Embedded Computational Elements in Extensible Routers

Scott C. Karlin

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

January 2003

Abstract

The demand to extend the set of services, such as network address translation, firewalls, proxies, and virtual private networks, that are supported by Internet-connected devices represents an opportunity to extend the traditional domain of Internet routers beyond simple packet forwarding. An important characteristic is the ability for end-users to install custom services on their routers. Routers with this characteristic are extensible.

Due to their critical position in the Internet topology, routers must be robust—when presented with unanticipated workloads, they must allocate their resources across the services they support according to administrator-established policies to ensure that each service gets the resources it needs.

By their nature, hardware-based routers with physically isolated control and data planes are robust but not readily extensible without a redesign while software-based routers may be extensible but are not robust without extensive regression testing; it is difficult to be simultaneously robust and extensible. The most common approach for router vendors is to favor robustness, and support new services on a case-by-case basis. Allowing the end-user to develop and install router services dooms this case-by-case approach to extensibility.

Emerging hardware in the form of intelligent, multi-port line cards that have their own embedded processing capabilities, based on either microprocessors or network processors, suggests that one can build cost-effective PC-based routers that lie in the design space between purely hardware- and software-based. However, the increased diversity of configurations makes both extensibility and robustness challenging. We do not want to require developers to re-implement services for every possible hardware configuration. How do we map the desired services onto the hardware to preserve robustness?

In this thesis we demonstrate that one can build a router from PC-based components, including programmable line cards, that is simultaneously extensible and robust. To show this, we describe an architecture, called VERA, that supports extensibility through an explicit interface and robustness through isolation of services; we present techniques to implement this architecture on a PC-based router; and we characterize and analyze the problem of mapping the services to the various, heterogeneous processors comprising the router to preserve robustness.

Acknowledgments

This dissertation represents a significant milestone in my life. I would not have reached this point without the support, guidance, and encouragement of many people along the way. While I cannot name everyone, I would like to acknowledge a few of them here.

I would like to begin by thanking my committee: Larry Peterson, Doug Clark, Kai Li, Brian Kernighan, and Randy Wang. As primary readers, Doug and Kai provided invaluable feedback that helped improve the clarity and focus of my dissertation. I have been very fortunate to have Larry as my thesis advisor; he has provided me with just the right mix of patience, encouragement, and feedback for my ideas. Additional thanks to Doug Clark who, along with Margaret Martonosi, advised me during my first two years at Princeton while I developed the SurfBoard as part of the Shrimp project led by Kai Li.

Thanks to Melissa Lawson, the graduate coordinator, for helping me negotiate the academic requirements and keeping me on track. Thanks to Trish Killian who placed dozens of orders for parts during the fabrication of the SurfBoard. Also, thanks to Jim Roberts and his technical staff for their support. In particular, I appreciate the time that Steve Elgersma and Chris Miller spent teaching me many of the finer points of system and network administration.

I have been privileged to meet and work with many wonderful colleagues including Andy Bavier, Zuki Gottlieb, Björn Knutsson, Aki Nakao, Patrick Min, Xiaohu Qie, Nadia Shalaby, Tammo Spalink, Dan Wang, Limin Wang, Mike Wawrzoniak, and Yuanyuan Zhou. Tammo and Zuki developed much of the IXP1200 microcode used in this dissertation. It was Björn (and then Andy) who convinced me that I should visit Sweden someday. I remember fondly the many hours Dan and I spent during our first semester working through the challenging but throughly rewarding assignments in Bob Sedgewick's Analysis of Algorithms class. I would like to thank my past and present office mates Angelos Bilas, Han Chen, Nitan Garg, Ting Liu, Anastasios Viglas, Lisa Worthington, and Peter Yianilos for letting me take more than my fair share of the bookcase space. Thanks also to Princeton undergraduates Jared Kramer, Mike Lindahl, Alexandra Blasgen, Marla Conley, Natalie Deffenbaugh, and Jennifer Borghi who have brightened my day (December 14th — give or take) each year since 1998. As an intern at Sarnoff, it was my pleasure to work with Ron Minnich, Maya Gokhale, John DeGood, Jim Kaba, and Aaron Marks with whom I had many interesting technical discussions. In addition, thanks to Jim and Aaron for introducing me to the joys of sushi.

My parents, Vern and Susan, deserve special thanks for encouraging and supporting me throughout my entire education. Also, thanks to my grandparents and my sister, Kristin, for their love and support. I would like to thank those who, through their mentorship and guidance, helped me find my way to Princeton: Frank Short, my first mentor; Lyn Hardy, my lab manager at TRW; and Ray Toal, my advisor when I was a Master's student at Loyola Marymount University. Thanks to my longtime friend Ed Felten for suggesting that I apply to Princeton.

Finally, this work would not have been possible without the support and patience of my wife Rosie and our daughter Molly. Rosie made a leap of faith when she agreed to move from her native Southern California to New Jersey. Once here, our years have been enriched by the friendship and support of many families including the DiLouie, Ildiz, Mapelli, McGinley, O'Connell, and Sheehan families. Thank you all.

This work has been supported in part by NSF grant ANI–9906704, DARPA contract F30602–00–2–0561, and Intel Corporation. During academic year 1999–2000, I was supported by an Intel Foundation Graduate Fellowship. Portions of this work were originally published in Karlin and Peterson [32].

To my family, who made this possible:

To my wife, Rosie, for her love and support and the courage to make the journey,

To my daughter, Molly, for the joy she brings to my life, and

To my parents, Vern and Susan, for their support and love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Few research artifacts have made as big an impact on society as the Internet. Both the number of connected hosts and daily users now number in the millions. As more users get connected, more people are developing novel ways of using this global resource. *Routers* are a key component of the Internet. They tie individual networks together and give the illusion that all the hosts on all the networks are directly connected. Because routers are the "glue" which binds the Internet together, they are in a unique position to modify, redirect, or monitor the data passing through them. In this thesis we demonstrate that we can build a router from PC-based components, including programmable line cards, that is simultaneously extensible and robust. To show this, we (1) describe an architecture, called VERA, that supports extensibility through an explicit interface and robustness through isolation of services; (2) present techniques to implement this architecture on a PC-based router; and (3) characterize and analyze the problem of mapping the services to the various, heterogeneous processors comprising the router to preserve robustness.

1.1 IP Routers

The Internet is a packet-based, store-and-forward, computer-to-computer communication network based on the *internet protocol* (IP) defined in RFC791 [49]. Rather than creating a dedicated channel from source to destination, transmitted data is transferred from device to device. At each device, the data is received, stored, and then forwarded to the next closest device along a path toward the destination in a technique known as *store-and-forward*. To improve the efficiency of the process, transmitted data is broken into fixed-size chunks call *packets* which are sent from device to device in a pipelined fashion.

1.1.1 Basic Router Functions

To support packet routing, IP specifies that every network interface have a distinct 32-bit identifier known as its *IP address*. Devices with more than one network interface have more than one address. To send data across the Internet, a host divides the data into blocks which will form the data portion of the packets. For each block, the host attaches routing information in the form of an *IP header* (see Figure 1.2 on page 5) to form an IP packet. Each packet is then encapsulated into a link-layer packet appropriate for the network interface (e.g., an Ethernet frame). Finally, the link-layer packet is sent to the router on the local network.

As depicted in Figure 1.1, an IP router has multiple ports (network interfaces) and a (general purpose) control processor connected with an internal switch. The complete specification for an Internet router is found in RFC1812 [2]; the two primary routing functions are:

Forward Packets: This is the obvious job of a router—moving packets arriving on one port and sending them out on another port so that the packets eventually reach

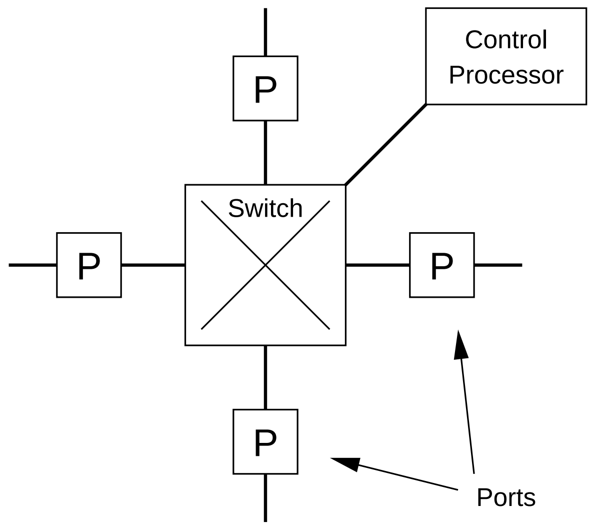

Figure 1.1: An Internet router consists of multiple ports (connecting to distinct networks), a control processor, and a switch that connects the components together.

their destination. To forward an incoming packet, the router must examine the header. By using the destination address as an index into its *routing table*, the router can then send the packet to the appropriate *next-hop* device (either another router or the destination host) on the network connected to the appropriate output port. Forwarding packets is considered part of the *data plane* of a router.

Maintain Routing Table: Because hosts and routers can join or leave the Internet at any time, routers must become aware of changes that effect their local routing tables. To support this dynamic aspect of the Internet, routers implement *routing protocols* to share connectivity information and maintain routing tables. The two primary routing algorithms currently in use are OSPF (Open Shortest Path First) [42] and BGP (Border Gateway Protocol) [53]. While the details and application of these protocols are beyond the scope of this thesis, the important point is that these protocols are sufficiently complex that reasonable implementations require the capabilities of the general purpose control processor. Maintaining the routing table is considered part of the *control plane* of a router.

One distinction between the data and control planes is that the former must process packets at line speed, while the latter is expected to receive far fewer packets (e.g., whenever routes change or new connections are established). The requirement that the data plane runs at line speed is based on the need to receive and classify packets as fast as they arrive, so as to avoid the possibility of priority inversion: i.e., not being able to receive important packets due to a high arrival rate of less important packets. The expectation that the control plane sees significantly fewer packets is only an assumption. It is possible to attack a router by sending it a heavier load of control packets than it is engineered to accept.

1.1.2 Minimal Packet Processing

All forwarded packets undergo some amount of transformation when they pass through a router. We refer to the minimum amount of processing (the most common case) as *IP--* [1]. Referring to the IP header depicted in Figure 1.2, the minimum amount of processing is to decrement the *time-to-live* (TTL) field and recompute the header checksum. Most packets only require IP-- processing. However, if the packet header has options, is destined for the router itself (e.g., a routing protocol packet), or is otherwise "exceptional," the router performs more processing on that packet than just IP--.

Because an important comparison metric for router manufacturers is the raw forwarding rate of packets only requiring IP-- processing (traditionally, the vast majority of packets), the path that these packets take through the router is often highly optimized. Such

[1] The "--" in the name refers to the decrement operator of the C programming language and is a play on the name of the C++ programming language.

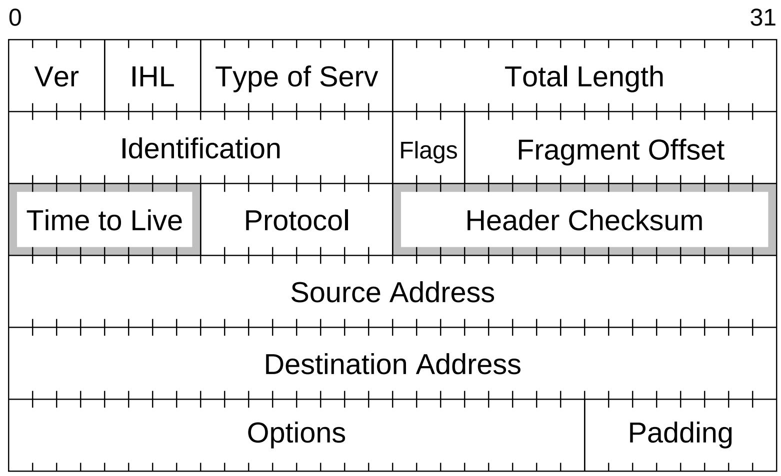

Figure 1.2: Selected fields from the Internet Protocol (IP) header. The tick marks denote bit positions. Only the shaded fields are modified as part of IP-- processing. The options and padding fields are optional. From RFC791 [49].

an optimized forwarding path is referred to as the *fast path*. Non-optimized forwarding paths are referred to as *slow paths*.

1.1.3 Traditional Router Implementation

One may characterize router architectures by their mapping of control plane and data plane functions to the underlying hardware. A *software-based* router is characterized by a single processor implementing both the data plane and the control plane. All packets in a software-based router are handled by the processor under software control. The design architecture is similar to (and often implemented with) a PC. That is, it contains a microprocessor, a shared bus (implementing the switch), and multiple *line cards* (implementing the ports). The actual hardware may be packaged in a desktop PC or as a stand-alone, custom-engineered device. Contemporary, open-source, operating systems such as Linux include the necessary software to implement a software based router on a PC.

A *hardware-based* router is characterized by an optimized fast path implementing all or part of the data plane using custom hardware (e.g., ASICs). The block diagram of hardware-based routers closely resembles Figure 1.1. The switch may be implemented as a high-bandwidth cross-bar switch rather than a bus. The line cards perform the route lookup and IP-- processing in custom hardware. Any portion of the data plane not implemented in custom hardware is handled by the control processor. Because the throughput of the custom hardware is significantly faster than the throughput of the control processor, hardware-based routers often have a large difference in throughput between the fast path and the slow path. Their overall performance is sensitive to the number of packets which cannot be handled by the fast path.

When comparing traditional router implementations one finds that software-based routers are generally less expensive than hardware-based routers. This is because software-based routers are often built from *commercial off-the-shelf* (COTS) components while hardware-based routers require components that are produced on smaller manufacturing scales. In general, hardware-based routers have higher-performance than software-based routers. However, this comparison is only true under the assumption that the vast majority of the packets require processing (typically, IP--) that can be handled by the optimized hardware data path.

1.2 Motivation

1.2.1 Internet Services

When the Internet was relatively small and primarily used by researchers in an open setting, the primary “service” was a point-to-point connection between two specific hosts allowing users to transfer files and interactively connect to remote hosts. Packets were treated equally and, other than IP-- processing, *independently* forwarded without modification from one specific host to another. While this is still the default behavior for IP packets, the shift in focus from an open research network to a vehicle for commerce, banking, telecommuting, and personal communication, along with a tremendous increase in its size, has led to the development and deployment of new services that require routers to forward and process packets in a variety of ways. These services include:

Firewalls: These devices are used to block the flow of packets between networks. For example, when placed at the boundary between a company and the rest of the Internet, its job might be to block external access to internal services and to block packets containing sensitive information from reaching destinations outside the company. This may be as simple as rejecting packets originating from a particular host to as complex as rejecting all the packets comprising an electronic mail message containing executable code matching the characteristics of a known computer virus. One of the benefits of quickly filtering out undesired packets is that it helps to reduce the effects of a *denial-of-service* (DoS) attack where devices under attack waste so much time handling spurious packets that legitimate packets are not handled in a timely manner. Since routers must examine each packet to determine its disposition and are already required to discard malformed packets, they can be easily extended to support a firewall service by including patterns in the route table for packets that should be explicitly blocked.

Intrusion Detection: By monitoring and analyzing packet traffic patterns and characteristics, intrusion detection services help determine when systems are under attack or have been compromised. A router can be a component of an intrusion detection system by passively monitoring the packets it receives and forwards. The router could perform the analysis itself, or redirect the collected information to a separate host for offline analysis.

Content Delivery Networks (CDNs): As the Internet became larger (both in the number of connected hosts and in its global reach), it became inefficient for large content providers, such as news organizations and multinational companies, to have a single web server with a single network connection provide content for every user. *Content delivery networks* (CDNs) consist of multiple *mirrors* (servers with a copy of some master data set) spread around the Internet. Specialized routers intercept server requests and transparently redirect them to a nearby mirror. This spreads the load across many devices and reduces the venerability of the service to single-point failures.

Network Address Translation (NAT): Because there is a limited number[2] of IP addresses available, NAT was developed to allow multiple devices on an internal, hidden network to appear as a single host (i.e., using a single IP address) on the Internet. Routers supporting NAT actively modify the source address of outbound packets and the destination address of inbound packet to maintain the illusion that the hidden hosts are directly connected to the Internet.

Commerce Servers: To support many simultaneous connections and to provide a measure of fault tolerance, Internet commerce sites often use a load balancing front-end router to dynamically direct traffic to the most lightly loaded server from in a set of

[2]While IP addresses are 32-bits wide giving more than 4 billion possible combinations, many are reserved by the protocol. In addition, due to the way blocks of addresses are allocated, it is not uncommon to have localized address space shortages.

servers (also called a *server farm*). Additionally, the front-end router could encrypt sensitive transaction data (e.g., credit card numbers and passwords).

Overlay Networks / Tunneling: Prior to widely available Internet connections, larger companies wishing to connect multiple locations were forced to create their own private networks using, for example, leased telephone lines. With the advent of low-cost, wide-spread Internet service, companies are now able to create *virtual private networks* (VPNs) that tie a subset of Internet hosts (those belonging to the company) together using encrypted *tunnels*. Routers implementing an overlay network tunnel packets through the Internet by treating the entire packet (including its header) as data, encrypting the data, and *encapsulating* the encrypted data into a new, larger packet. This "outer" packet is sent over the Internet in the regular way to another host in the VPN. Upon arrival, the packet is "unwrapped," decrypted, and delivered to its destination. Note that the destination may be the host itself or a locally connected machine. In some cases, the packet is forwarded to another node in the VPN by re-encrypting and re-encapsulating the packet. A VPN is an example of an *overlay network*. While end-hosts often perform the tunneling, routers are well-placed to make optimizations based on their innate knowledge of the underlying network topology.

Transcoding Media Gateways: With the proliferation of Internet-connected devices including cellular telephones, 2-way pagers, and personal digital assistants (PDAs), there is a market for efficiently delivering web-based content to these devices. A *transcoding media gateway* is a specialized router that converts high-bandwidth or high-resolution data (e.g., video [11, 19]) to lower-bandwidth (suitable for wireless links) or lower-resolution (suitable for handheld displays). Thinning a data stream to match the capabilities of the end device can reduce latency and improve efficiency in the use of transmission bandwidth.

Quality of Service: While not a service itself, *quality of service* (QoS) introduces a range of packet priorities. In a traditional router, packets are independently processed in first-in, first-out (FIFO) order on a per port basis and are forwarded using *best effort* processing—that is, packets are treated equally and every effort is made to forward them. It is easy to imagine instances where some packets are more important than others and require priority processing. A common way of generating additional revenue from a service (such as providing Internet connectivity) is through service differentiation. By varying the quality of service, *Internet service providers* (ISPs) can charge customers more to have their packets be given priority potentially increasing the ISP's revenue.

The result is that routers that can (1) perform more complex decisions than simply forwarding packets based only on their destination address, (2) perform more complex packet processing than IP--, and (3) perform service differentiation rather than use best-effort packet delivery will be able to distinguish themselves in the marketplace. Note that one might view this situation as a reason to give servers multiple ports and have them perform more router-like operations. Either way, the point is that there is now a continuum of devices from the pure router to the pure server. This thesis chooses the router-centric view by focusing on *extensible routers*—routers which are designed to support the end-user installation of new services such as those described above.

For hardware-based routers, additional services mean that fewer packets (e.g., IP--) can be handled without requiring the resources of the control processor; said another way, there are more "exceptional" packets. For both software-based and hardware-based routers, these “exceptional” packets require more cycles per packet as the number and complexity of the Internet services increases. This means that for a fixed architecture, an increase in the fraction of “exceptional” packets is an increase in the fraction of the packets will need to be processed by the slow path. The reason packets move to the slow path is simply that the fast path does not have the capability to process these exceptional packets—the programmable cycles implementing the service are too far away. The next section presents emerging hardware that suggest an approach to this problem.

1.2.2 Emerging Hardware

Without even considering advances in networking technology that enable the high-speed transmission of bits, emerging hardware is influencing the design of routers. At the chip level, we have seen a steady increase in the speed and capabilities of programmable devices including microprocessors and field-programmable gate arrays (FPGAs—devices whose internal logic and interconnections can be quickly re-configured with different functionality) and the introduction of special purpose, high-speed, fixed-function processors such as MPEG video encoders/decoders and cryptographic processors.

As the performance of programmable devices increases, there is an opportunity to replace complex, fixed-function logic and state machines with software running on commercial off-the-shelf microprocessors. This makes sense—the inherent flexibility of software along with its ability to be changed after (or during) the hardware manufacturing phase makes it an attractive option for a growing number of devices. Of particular interest is the introduction of processor-based line cards for PCs because routers based on PCs [63] have the advantage of leveraging the economies of scale of the PC industry. The first generation of processor-based line cards simply replace fixed-function logic and, in some cases, provide limited, hard-coded, protocol support (e.g., checksum computation) to offload some specific computation from the host processor. These line cards are not intended to run customer software. To minimize production costs, the processor speeds are carefully engineered to provide the minimum number of cycles needed to meet the board's specifications.

The second generation of processor-based line cards use general-purpose microprocessors and have the ability to run end-user software. Examples include a MIPS-based card from Alteon [1] and the PMC694, a PowerPC-based card from RAMiX [52]. The number of processing cycles that can be applied to a packet depends on the relative speed of the processor and the aggregate bandwidth of the ports. The Alteon card (marketed as a server network card intended to ease the host processor burden by offloading some low-level protocol processing from coalescing multiple interrupts) has a single, full-duplex, 1 Gbps port and two MIPS processors each running at 88 MHz giving a ratio of 0.09 cycles/bit. The PMC694 card (designed to handle all packet processing and give Internet connectivity to processor boards in industrial and military applications) has two, full-duplex, 100 Mbps ports and a single 266 MHz PowerPC processor giving a ratio of 0.7 cycles/bit. Based on this metric, the PMC694 can apply approximately eight times as many processing cycles to each packet as the Alteon card.

Figure 1.3 shows an example 4-port router using two PMC694s in a Pentium-based PC. While this router configuration has relatively few ports, it has two especially interesting features: (1) a high processor cycle to port bandwidth ratio, and (2) substantial processing cycles "close" (low-latency) to the ports.

The current third-generation processor-based line cards are based on *network processors*. A network processor is a programmable device specialized to handle packets at high speed. They are characterized by multiple, independent processing elements (often

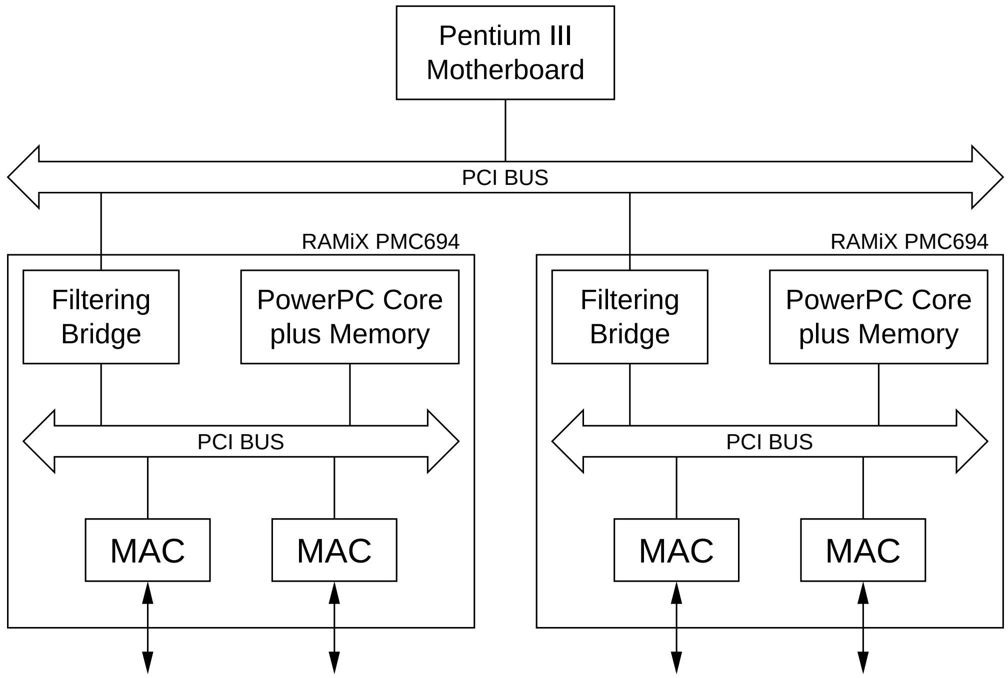

Figure 1.3: A PC-based router using PowerPC-based programmable line cards with a total of four 100 Mbps ports.

called *engines*) and multiple buses allowing packets to move efficiently between network interfaces and buffer memory. Network processors have been introduced by Vitesse [60], Intel [27], and IBM [24] among others.

As an example of the kind of hardware configuration that is possible, Figure 1.4 shows an example 8-port router consisting of a Pentium processor, connected by its PCI bus to an Intel IXP1200 network processor. The IXP1200, in turn, is connected by a proprietary IX bus to eight 100 Mbps Ethernet ports. Internally, the IXP1200 consists of a StrongARM processor, plus an array of six microengines.

The recent availability of high-performance, multi-port line cards for PC systems based on programmable (either microprocessor or network processor) devices creates the problem of creating a router architecture that can support a broad range of configurations.

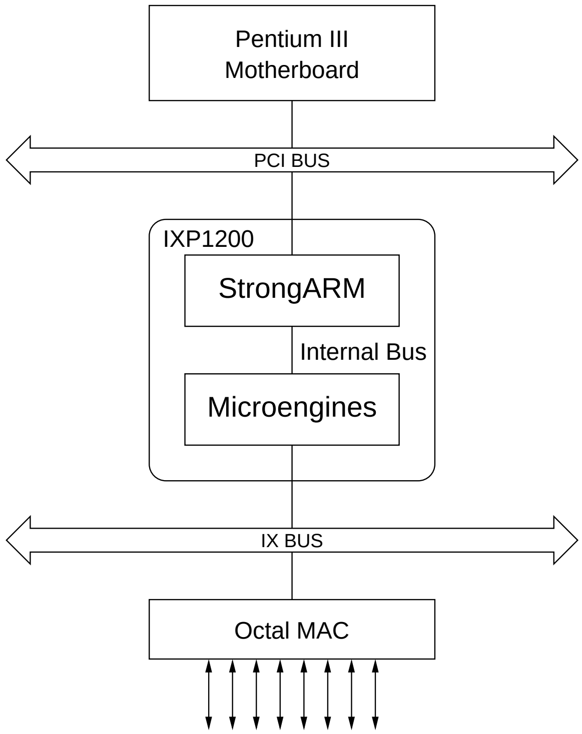

Figure 1.4: A PC-based router using an IXP1200-based programmable line card with a total of eight 100 Mbps ports.

1.3 Problem

The problem this thesis addresses is how to build a router using commercially available, PC-based components that is both extensible and robust.

1.3.1 Key Definitions

Because extensibility and robustness are key elements of our thesis, we define them here.

Extensibility

An extensible router is one that is programmable by the user, where "user" is defined as someone other than the original equipment manufacturer. For example, a user might be the system administrator for an Internet service provider (ISP) or a third-party value-added reseller (VAR) making custom routers for a niche market. An extensible router is programmable in the sense that when a packet arrives, user code can operate on the bytes of the packet. As a router, an extensible router has intrinsic support for processing packets. Note that it is beyond the scope of this thesis to state how easy or difficult it is to add new services or precisely how code is structured except how it influences robustness.

Robustness

The definition of robustness assumes that the administrator or user of the router will want to allocate its resources differently for different classes of packets. Even traditional routers need this property. For example, routing protocols often use "are-you-alive?" queries to neighboring routers. If a nearby router does not receive a timely response, it may assume that the router is inoperative and attempt to route packets around the trouble. Therefore, even if a router is receiving an unanticipated flood of low-priority packets, we still want it to process and respond to these high-priority administrative packets.

We want the administrator to allocate the resources to the various services so that individual services are guaranteed to have enough resources or are prevented from using too many resources. The administrator makes these allocations based on an anticipated workload. A robust router is able to follow these allocations for any workload. Note that a robust router must limit the administrator to making allocations that the router can guarantee. In summary, robustness has two parts:

(1) A robust router must be able to read and classify packets at line speed.

(2) A robust router must honor the processing guarantees it makes.

Note that the classification performed in part (1) need only be as deep as needed to honor the guarantees it makes in part (2). One guarantee routers typically make is to forward IP-- packets at line speed.

1.3.2 Service / Flow / Packet Model

Central to the concept of routers providing services is that of a packet *flow*. A flow is defined as a set of packets that arrive on a given port and receive the same treatment by the classifier associated with the input port. A flow is characterized by the triple, ⟨input port, forwarder, per-flow state pointer⟩. A *service* is a collection of one or more related flows. Corresponding to each service is *per-service state* that can be manipulated in the same way as per-flow state.

Every packet belongs to a flow; every flow is part of a service. For example, every router implements the IP-- service with separate forwarding paths to handle flows between every pair of ports. As another example, to implement a content distribution service, a flow might correspond to all packets addressed to TCP port 80, and the forwarding function might be a redirector that edits the packet header. As a final example, to implement an adaptive video streaming service, one might define a flow to be all packets between a particular pair of UDP ports, and the forwarding function might selectively drop packets (video layers) that exceed the available network capacity [11].

The keys to providing QoS in a router are, first, to track resource commitments and only admit services that will not overextend any given resource; second, to verify that services are not exceeding their resource profiles; and third, to ensure that packets that are not part of any service, for example, those from a DoS attack, do not consume enough resources to prevent the router from otherwise meeting its commitments. By separating flows from services, we can enforce limits and maintain state on both a per-flow and a per-service basis.

1.3.3 Discussion

As discussed in Section 1.1.3, traditional routers can be characterized as being hardware based or software based. Faced with goals of robustness and extensibility, both traditional approaches fall short. Hardware based routers are generally robust. There is a natural separation of the data plane (implemented in hardware) and the control plane (implemented in software on a general purpose processor). This separation (or parallelism) allows the router to classify (and usually forward) most packets without impacting any concurrent processing occurring in the control processor. But because the data plane is implemented in hardware, the only way to support extensible router services on a traditional hardware based router is to implement the services on the control processor. In some cases this may be feasible; however, the small data path from the switch to the control processor typical of traditional routers will quickly saturate as more packets require processing that cannot be done on the hardware data plane.

By their nature, software based routers are better suited to supporting extensions. However, because the processor must implement both the data plane functions as well as the control plane functions, the software must be carefully crafted and tested to ensure that the router is robust. Router vendors need three months or more [12] to perform regression testing on software based routers when changes are made to the software. Even changes to the user interface code may impact the forwarding performance of a router.

Therefore, traditional hardware-based routers are robust but not extensible while traditional software-based routers are extensible but not robust; it is difficult to be both at the same time. The most common approach is to favor robustness, and address extensibility on a case-by-case basis. That is, the router manufacturer redesigns the hardware or software as each new service is supported. However, the increased demand for new router services (cf. Section 1.2.1) dooms the case-by-case approach to extensibility.

In addition, the increased availability of network processors and intelligent line cards (cf. Section 1.2.2) is a double-edged sword. On the one hand, network processors represent an opportunity to find a middle-ground between software-based and hardware-based routers, thereby supporting both extensibility and robustness. On the other hand, there is the problem of dealing with an increasingly diverse set of hardware configurations. This makes both extensibility hard (one would hate to have to re-implement services for every possible hardware configuration) and robustness hard (how does one map the desired services onto the hardware in such a way that robustness is preserved?). Supporting extensibility while maintaining robustness is precisely the problem we address in this thesis. Before outlining our approach in Section 1.4, we discuss related work to put our contribution in context.

1.3.4 Related Work

Much work has done in the area of router design, but none solves the problem of extensibility and robustness using commercially-available PC-based components. Many currently available general-purpose operating systems can be configured to route Internet packets. However, as general-purpose systems, they were not necessarily designed with packet forwarding in mind. For example, we have measured Linux to be up to six times slower forwarding IP-- packets than Scout [51]. More importantly, however, such general-purpose systems provide no explicit support for adding new forwarders. Unfortunately, router operating systems in commercial products are closed, and the extent to which they are extensible is not clear.

Other research efforts recognize the issue of extensibility, but focus on how to structure software-based routers so they can be easily extended/modified, but without considering robustness. For example, recent systems like Click [34] and Router Plugins [13] do support extensibility, but consider neither distributed heterogeneous processing environments of a PC with intelligent line cards nor the issue of robustness. Other efforts to define architectures for active networks [23, 64] also support extensibility, but pay little attention to either performance or robustness. In contrast, our focus is on how to make a robust IP router extensible.

The Dynamically Extensible Router (DER) [36] project addresses both extensibility and high-performance. However, the DER effort is focused on the design of custom hardware and, therefore, cannot benefit from the economies of scale of the PC industry.

Walton, *et al.* [63] describe how to make efficient use of the PCI bus on a PC-based router by exploiting peer-to-peer DMA. We take advantage of the same technique, but extend their work in several ways. For example, we use intelligent line cards with much higher processor cycle to port bandwidth ratios than they consider; we assume the possibility of moving the functions that implement the user-defined services onto the line cards. We also pay particular attention to how the various resources (bus, CPU) are scheduled, and the synchronization problems that occur when multiple source devices try to access the same destination device.

The Suez [8, 50] project is an effort to build a router from a cluster of PCs. It can be considered a system similar to SHRIMP [5] that is specifically targeted to the task of routing. Because Suez uses single port LANai-based Myrinet cards [6] and all packets arriving on an interface must traverse the PCI bus, the bottleneck in this system is the PCI bus. We will show in Chapter 4 that having multiple ports on a card, allowing services to be installed on the line card, and only sending the header across the bus (when feasible) reduces the PCI traffic to the point that it is not the bottleneck in the systems we modeled.

The SPINE project at the University of Washington [10, 17] addresses operating system issues to support intelligent line cards. Like Suez and SHRIMP, they focus on line cards (in their case, the LANai processor on Myrinet cards) with a single port.

Programmable network cards have been used for a number of purposes over the years, including to provide access to high-speed links [15, 59, 61], improve handling of multimedia streams [16], and implement distributed shared memory. All of these prior efforts have been limited to end hosts (rather than routers), and the line cards support a single network port.

Recent work by Lepe-Aldama and García-Vidal [38, 39] at the Polytechnical University of Catalonia on PC-based software routers directly addresses PCI resource allocation. However, they only consider a single PC (with a single CPU) using line cards without processing capability. They also assume the existence of line cards which have an independently operating DMA channel per *flow*. Because routers must support hundreds of flows, we do not feel that such an assumption is reasonable. As a point of comparison, the line cards we will consider only have two DMA channels each.

Several recent projects have also focused on the problem of making it easier to extend router functionality [13, 34, 50], but to date these have been limited to single-processor Pentium-based implementations. The exception is a recent effort at Washington University to study the feasibility of implementing router extensions in FPGAs [58]. Perhaps the work closest to our own is an ongoing effort to port the Genesis kernel [7, 35] to the IXP1200. Genesis is designed to support virtual networks by dynamically loading routelets (similar to our forwarders) onto the IXP1200. The main difference is that our approach runs all forwarders for a given packet in a single thread, which is critical to our ability to isolate performance under varying loads.

1.4 Thesis Contribution

As discussed in the previous section, the expanding array of services breaks the case-by-case approach to supporting these services. We therefore need a router that is designed to quickly support new services—an extensible router. At the same time, we need the router to continue to be robust. Finally, the problem such a large space of hardware configurations creates is one of mapping the functions that implement the services onto particular processors—and by implication defining the switching path for the packets—onto a particular hardware configuration. Our approach to this problem is to define a virtual router architecture. The main contribution of this thesis is to demonstrate that we can build a router from commercially available, PC-based components (specifically, multi-port, programmable line cards) that is simultaneously extensible and robust. To show this, we (1) describe an architecture, called VERA, that is extensible and robust; (2) present implementation techniques to realize this architecture on a PC-based router; and (3) characterize and analyze the problem of mapping the functions implementing the router services to the various, heterogeneous processors comprising the router in a way to preserve robustness.

1.4.1 VERA

Figure 1.5 shows how the VERA framework constrains and abstracts the essence of both the routing function space and the hardware configuration space. VERA consists of a router abstraction, a hardware abstraction, and a distributed router operating system. The router abstraction must be rich enough to support the RFC1812 requirements as well as a variety of extensions. The hardware abstraction must be rich enough to support a variety of hardware components. However, it should expose only enough of the hardware details needed to allow for efficient router implementations. Note that both abstractions must be well "matched" to one another so that the map between them (i.e., the distributed router operating system implementation) is efficient and clean. The abstractions must also be chosen to allow us to model and reason about the system with adequate fidelity without also getting bogged down by details. Our approach to VERA is guided by three goals:

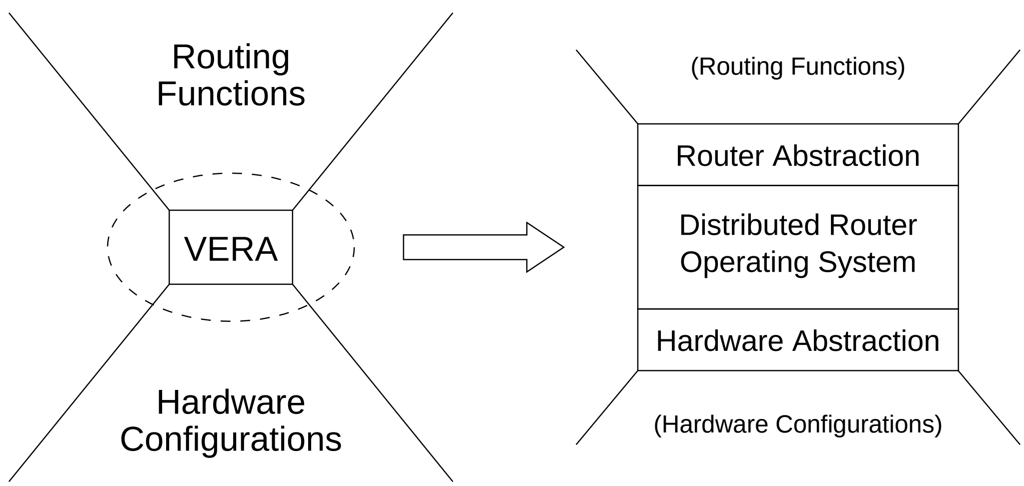

Figure 1.5: VERA constrains the space of routing function implementations and hardware exposure to facilitate the mapping between the two.

Extensibility: Our design must export an interface and protocol that allows new functionality to be easily added to the router. It should be easy for a trusted entity (specifically, the end user) to inject new functionality into the router, including both new control protocols and code that processes each packet forwarded through the data plane. The challenge in supporting extensibility is defining the interface by which the control program interacts with the code running in the data plane. A specific goal is to develop an architecture that is compliant with the router requirements of RFC1812 including broadcast and multicast.

Robustness: The router should continue to behave correctly regardless of the offered workload. That is, for every (new) service admitted by the router, the resource allocation set by the administrator continues to be enforced by the router. For example, it should not be possible to inject code into the data plane that keeps the router from processing packets at line speed, and likewise, it should not be possible for a high packet arrival rate to choke off the delivery of control packets to the control plane. Our approach is to isolate the interaction among the components so that an increased load on one component will not impair the performance of another component. The challenge is to create an architecture with this characteristic.

Efficiency/Performance: In addition to being extensible and robust, we want the architecture to support efficient implementations for a given hardware configuration. For example, by taking advantage of the processor on intelligent line cards, many packets can be completely processed and forwarded without involving the main processor at all. By offloading the main processor as much as possible, we leave extra headroom for user-level extensions. The router should be able to forward packets at the highest rate the hardware is able to support. The challenge is to simultaneously manage processing and switching resources across multiple, heterogeneous elements within the router.

In the development of VERA, we have made a series of design choices that, when taken together, provide a consistent and coherent framework. Nearly every design choice represents a trade-off among performance, complexity, and modularity. Because VERA is designed for implementing extensible IP routers on a heterogeneous processing environment based on commercial off-the-shelf hardware, we have made significantly different design choices than either a general-purpose operating system or a router operating system for a single, centralized processor. One of the main contributions of this thesis is to identify and motivate these design choices.

1.4.2 Implementation

During our implementation of portions of VERA using intelligent line cards in a commodity PC, we developed several implementation techniques for PC-based routers. These are discussed in Chapter 3 and include (1) a Linux device driver for the host processor which forms much of the hardware abstraction layer, (2) techniques for moving packets from device to device, and (3) techniques for indirectly managing the shared-bus resource.

1.4.3 Mapping

VERA is specifically designed to support multiple services with varying resource requirements on multi-processor platforms. As a result, a key issue is the mapping of services to the processors in the system. This placement problem also becomes one of admission control as we allow the possibility that new services may be dynamically installed on an extensible router. This thesis contributes an analysis of different placement strategies over a range of service workloads.

1.4.4 Thesis Outline

To demonstrate that one can build a router that is both extensible and robust, we begin by presenting an architecture (Chapter 2) that is extensible by design and supports robustness by isolating the user-installed services. Next, through the implementation of key parts of the architecture (Chapter 3), we show that an implementation on commercial off-the-shelf hardware can maintain robustness.

Note that the architecture and the implementation are only extensible if the router has enough available resources to support new services. In Chapter 3, we show two hardware configurations that have the available resources to be extensible and the isolation to be robust. Note that the architecture and implementation rely on the fact that the resource requirements of the services allowed to run on the router do not exceed the capabilities of the router. To ensure that the router is robust, there is an admission control decision when a user attempts to install a new service on the router. If the requirements of the service exceed the available resources of the router, the router cannot guarantee that it will be robust and the service is not admitted. However, if the service is admitted, there is an additional decision to place the service on the appropriate processor within the router. Our approach is to combine these two decisions as part of resource allocation (Chapter 4) for the router.

In summary, we first argue that our architecture is extensible and robust by design. We then show that this architecture can be implemented so that extensibility and robustness are preserved as long as the running system is not oversubscribed. Finally, we describe and evaluate our resource allocation mechanism that prevents a router from becoming oversubscribed.

Chapter 2

Architecture

A key architectural goal of VERA is to support a variety of user-installed router services on a range of hardware configurations in a manner that preserves the robustness of the router. This chapter defines the hardware abstraction and the router abstraction and provides an initial discussion of some of the issues the abstractions raise.

2.1 Hardware Abstraction

This section describes the hardware abstraction layer for VERA. The object of any hardware abstraction layer (HAL) is to define an interface between the hardware and the "device independent" upper level software (typically, an operating system). This allows us to support a variety of hardware configurations without rewriting the operating system. Choosing the appropriate level of abstraction is somewhat of an art. We want to choose the abstraction level so that everything below is part of a consistent and uniform HAL and nothing above directly accesses the hardware. If we select too high a level of abstraction, each port to a new hardware configuration will require a major effort. If we select too low a level of abstraction, we will not be able to take advantage of higher-level capabilities provided directly by the hardware without breaking through the abstraction layer.

2.1.1 Hardware Primitives

The hardware abstraction layer for VERA can be broken down into three major components: processors, ports, and switches. The abstractions for the ports and processors are fairly standard. The abstraction for the switching elements is more involved and represents the bulk of the hardware abstraction layer. We describe each of the abstractions here:

Processors: The hardware abstraction layer groups the actual processors into virtual processors. Each virtual processor is either a single processor, a symmetric multiprocessor (SMP), or a tightly coupled set of processing elements such as the six microengines in an IXP1200 network processor. The relevance is that each virtual processor is its own *scheduling domain* with a single thread pool. Also, any memory local to a processor is consolidated with and managed by that processor.

Ports: A device driver hides the register level details of the port hardware interface and provides a uniform software interface for upper layers. Even though a given port may be directly accessible by more than one processor, each port is *managed* by a particular processor. This assignment is static. The port interface exports a scatter/gather capability that can read and write the packet header and data from separate memory locations. Note that these memory locations must be local to the processor that manages the port.

Switches: The switching elements are modeled as passive (no processing cycles) and shared (all devices connected to a switch share a single bandwidth pool). This also means that we assume that there are no explicit control registers accessible to the software to schedule data movement through the switch. VERA's switch abstraction provides a *distributed queue* interface for interprocessor data movement. Distributed queues span switches; that is, they are queues whose head and tail are on different processors. This interface is needed by the distributed router operating system to implement the interprocessor communication and message passing—the basis for all the data movement within the router.

In addition to the abstractions of the processors, ports, and switches, the hardware abstraction maintains a static database containing the topology of the system, as well as the capabilities of the components. This is used in conjunction with the performance model and resource management schemes discussed in Chapter 4.

While our examples in this thesis are focused on single-PC routers with multiple intelligent line cards, VERA was also designed to apply to loosely coupled clusters of personal computers that utilize gigabit Ethernet, InfiniBand, or some other system area network (SAN) technology [6, 26] as the underlying switching technology. This will be revisited in Chapter 3.

2.1.2 Processor Hierarchy

Figures 2.1 and 2.2 show the hardware abstraction graph of the systems shown in Figures 1.3 and 1.4, respectively. The nodes of the graph are the hardware primitives: processors, ports, and switches. The solid lines connecting the nodes indicate *direct* connections where data can flow between primitives. A consequence of our requirement that each port be managed by a processor is that all packets arriving on a port must first take a direct path from the port to the managing processor and that all pack-

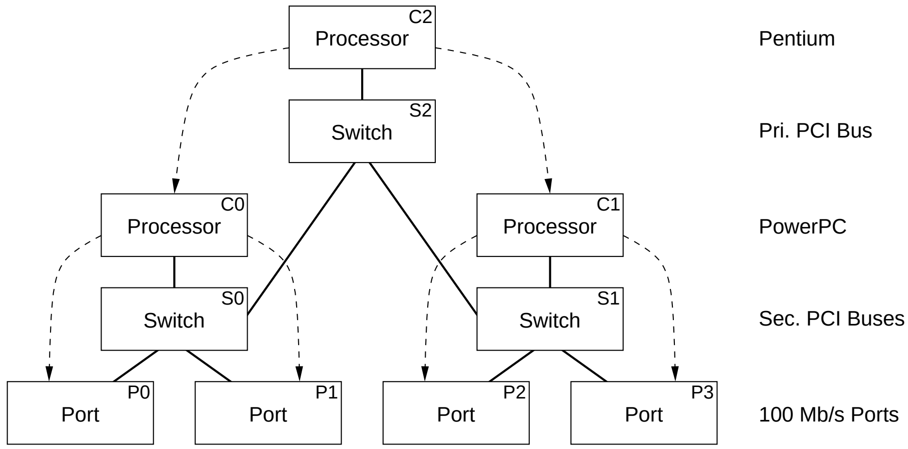

Figure 2.1: The hardware abstraction graph of Figure 1.3. The solid lines indicate packet flow paths. The dashed arrows show the processor hierarchy.

ets exiting on a port will ultimately take a direct path from the managing processor to the port. A *switching path* is the ordered sequence of hardware primitives traversed by a packet as it moves through the router. From the hardware abstraction graphs, one can enumerate the port-to-port switching paths. Representative switching paths for Figure 2.1 include: **P0–S0–C0–S0–P1**, **P0–S0–C0–S0–S2–C2–S2–S1–C1–S1–P2**, and **P0–S0–C0–S0–S2–S1–C1–S1–P2**. Representative switching paths for Figure 2.2 include: **P0–S0–C0–S0–P1**, **P0–S0–C0–S1–C1–S1–C0–S0–P1**, and **P0–S0–C0–S1–C1–S2–C2–S2–C1–S1–C0–S0–P1**. Note that Figure 2.1 has three distinct switching elements (rather than one) because each filtering bridge segregates its secondary PCI bus from the primary PCI bus buses and, thus partitions the bandwidth. Figure 2.1 also illustrates why we bother to make an assignment of each port to a processor; in this system, it is possible to configure the PCI bridge chips so that any of the processors can access (and therefore control) any of the ports. By not allowing each

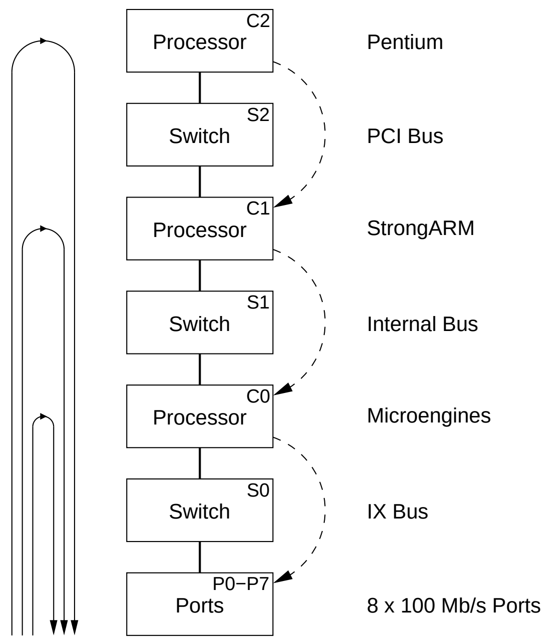

Figure 2.2: The hardware abstraction graph of Figure 1.4. The solid lines indicate packet flow paths. The dashed arrows show the processor hierarchy. The flow lines on the left indicate the three possible hardware paths.

processor direct access to every port, we are choosing a simpler, more modular, and more scalable architecture for one which is potentially faster but more complex.

By ignoring the switches and considering only the processors and ports, we can find a spanning tree with the *master processor* at the root and all the ports as leaves. This spanning tree is called the *processor hierarchy*. The dashed arrows of Figures 2.1 and 2.2 indicate the edges of the spanning tree defining the processor hierarchy. While not a requirement, we generally expect that as we move up the hierarchy (away from the ports), processors will have more general purpose cycles available. Consider, for example, Figure 2.2. At the lowest level, packets traverse only microengines, while at the highest level, packets are processed by the Pentium. An intermediate level corresponds to a

StrongARM processor on the IXP1200 chip. At each level of the hierarchy, the packet has access to some number of cycles, but there is overhead involved in reaching those cycles. Higher levels (e.g., the Pentium) offer more cycles, but packets also consume resources at lower levels of the hierarchy to access them. Lower levels (e.g., the microengines) have enough cycles to perform only certain operations at line speeds.

2.1.3 Hardware API

This section outlines some of the hardware layer API function calls that encapsulate the distributed queue functionality:

q* = allocQueue*(endpoint, dir, depth, parms)

This function allocates a distributed queue, *q*, connected to processor *endpoint*. The queue is configured to hold *depth* entries. The direction parameter, *dir*, can be set to either **`incoming`** or **`outgoing`**. The additional parameters include the switch reservation.

insert*(queue, item)*

This function inserts the *item* on the given *queue*.

item* = remove*(queue)

This function removes the *item* from the given *queue*.

2.1.4 Justification

Here we describe some of the dimensions of the space and explain what the choices we made for VERA are reasonable for extensible routers.

Communication Semantics: VERA uses a message passing model (as opposed to a shared memory model) which more closely models the packet flow in router hardware across a wide variety of devices. A message passing model is better suited for hardware configurations which include groups of PCs connected via Ethernet-based switches acting as a single router.

Command Structure: A hierarchical control structure is beneficial for two reasons. First, a single root processor simplifies the maintenance of routing tables. The alternative (a distributed control structure) would require a routing protocol to run *within* the router (e.g., between the processors in the router) wasting processor resources. Second, most *media access controller* (MAC) chips which implement the ports are designed with the assumption that there will be a single point of control. Allowing multiple processors access to a single port may not always be possible.

2.2 Router Abstraction

This section defines the router architecture visible to programmers. The main attributes of the architecture are that it hides details of the underlying hardware while providing a framework that supports the services described in Chapter 1.

2.2.1 Router Primitives

The router abstraction layer for VERA consists of three primary primitives: classifiers, forwarders, and output schedulers. These primitives are connected by queues to form the three-stage pipeline depicted in Figure 2.3 and capture the three main functional aspects of the router:

Classify: Each input port as an associated *classifier* that receives packets and sends them to the appropriate forwarder. Classifiers do not modify packets. To support multicast and broadcast, classifiers can "clone" packets and send them to multiple forwarders.

Forward: The *forwarder* gets packets from its single input queue, applies a *forwarding function* to modify the packet, and sends the modified packet to its single output queue. All transformations of packets in the router occur in forwarders. In addition to modifying the packet, the forwarder maintains state (later called *per-flow state*) that can be consulted and modified by the forwarding function. The classifier provides the forwarder with a pointer to the (per-flow) state.

Schedule: Each output port has an associated *output scheduler* that selects one of its non-empty input queues, and transmits the associated packet. The output scheduler performs no processing (including link-layer) on the packet. Note that output schedulers are different than the *thread schedulers* described in Chapter 3.

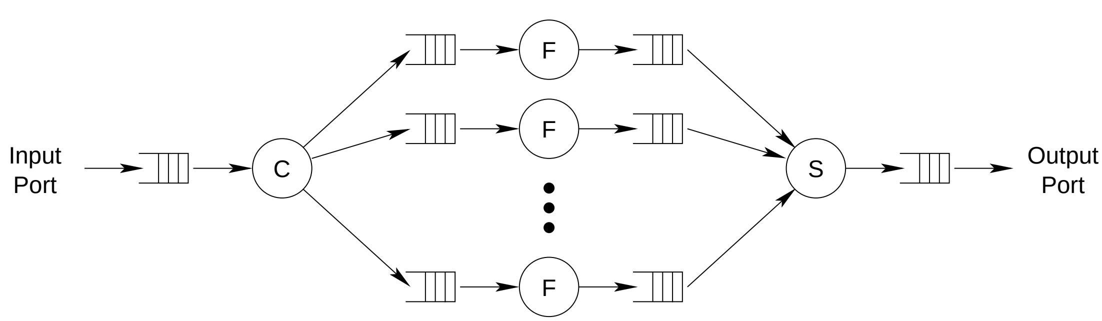

Figure 2.3: The classifying, forwarding, and scheduling of IP packets.

A path from an input port, through a classifier, through a forwarder, through an output scheduler, and out an output port is called an *abstract forwarding path*. The result of mapping an abstract forwarding path onto a switching path is called a *concrete forwarding path*. When the distinction is clear from the context, we use the term *forwarding path* and drop the "abstract" or "concrete" qualifier. Figure 2.4 illustrates mapping an abstract forwarding path onto the **P0–S0–C0–S0–S2–S1–C1–S1–P2** switching path of Figure 2.1. At router initialization time, each port has its associated classifier and scheduler and there is an initial set of pre-established forwarding paths connected to the classifiers and schedulers. To support QoS flows and extensions, our architecture allows forwarding paths to be dynamically created and removed by existing forwarders. In fact, the core architecture supports a generic forwarding infrastructure; even basic IP functionality is treated as an extension. Section 2.2.3 gives details about this application programmer interface (API).

2.2.2 Classification Hierarchy

Our router architecture recognizes that packet classification is not a one-step operation. Instead, we view classification occurring in distinct stages. A simple classification sequence might be:

Sanity Check: the first step of classification is to identify packets which must be ignored or are malformed. Packets that are not identified as malformed are sent to the next level.

Route Cache: at this level, the packet is quickly compared against a cache to determine the correct forwarder within the router. Packets not in the cache, as well as packets that require special processing, are sent to the next level.

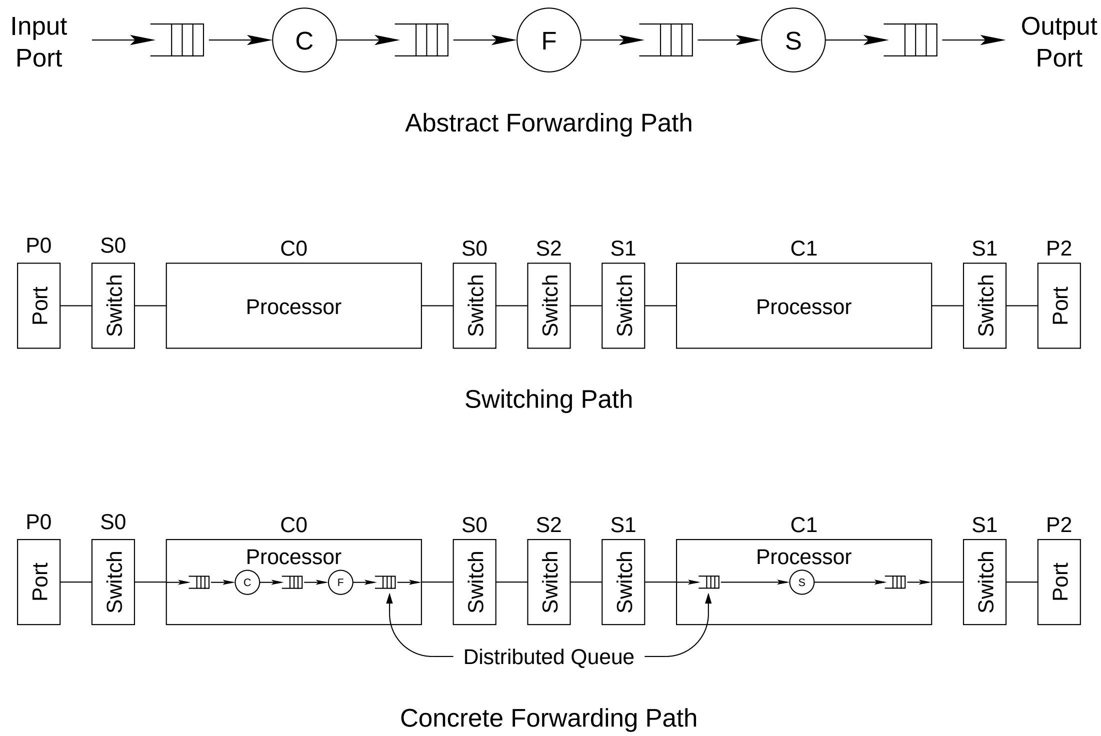

Figure 2.4: Mapping an abstract forwarding path onto a switching path.

Prefix Match: most routers run a prefix matching algorithm that maps packets based on some number of bits in the IP header, ranging from just the destination IP address to the source/destination addresses and ports [14, 37, 57, 62].

Full Classification: eventually, packets which have not been classified in early stages will reach a "mop-up" stage that handles all remaining cases, including application-level routing. This stage is often implemented with arbitrary C code.

Figure 2.5 shows that the internal structure of a classifier is really a hierarchy of subclassifiers. Once a packet leaves the classifier *C*, the packets are *fully* classified—a specific forwarder is the target.

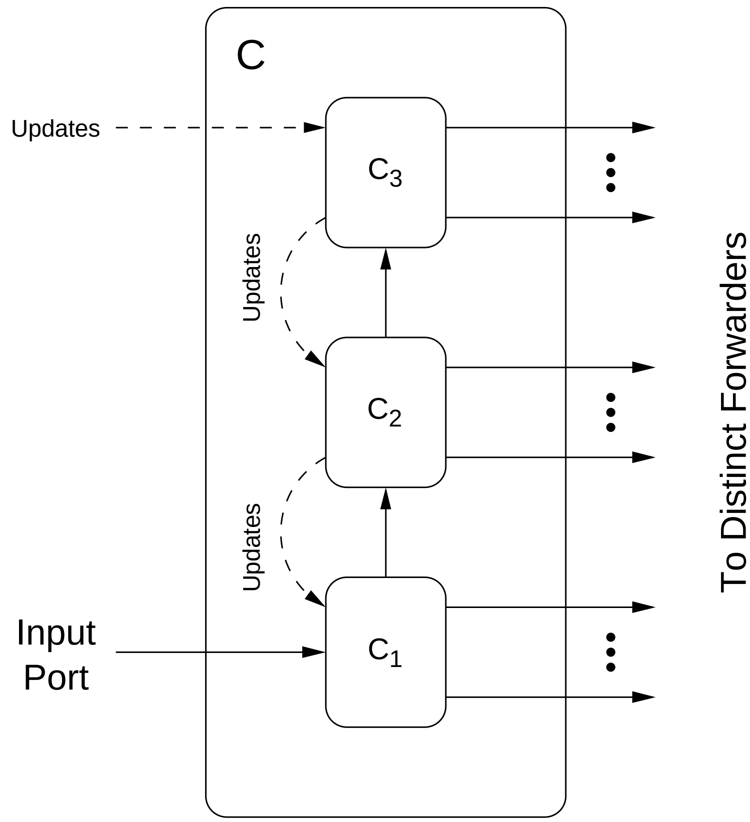

Figure 2.5: Classification Hierarchy. Classifier C is composed of partial classifiers C_1, C_2, and C_3. Solid lines indicate packet flow. Dashed lines indicate classification updates (e.g., route table updates).

For our architecture, we impose the restriction that the outputs from a classifier are unique. Referring to Figure 2.5, this would mean that an arrow from C_1 could not point to the same forwarder as an arrow from C_2, for example.

Although we believe that this hierarchical structure is a fundamental aspect to packet classification on an IP router [48], the exact form of the hierarchy is often dictated by the processor hierarchy onto which it is mapped. We return to this issue in Chapter 3.

One additional issue that arises from our experience has to do with managing the classification hierarchy. It is often the case that a packet can only be classified by an upper level of the hierarchy, meaning that we need to ensure that lower levels do not. For example, suppose C_i is designed to match layer-4 (i.e., transport layer) patterns and C_{i-1} holds a route cache. If we attach a new forwarder to the classifier that specifies a level-4 pattern, we need to update C_i to add the new layer-4 pattern and we also must ensure that there is a miss in the route cache at C_{i-1}. The right side of Figure 2.6 shows how a cache will obscure higher classification levels; any change to the tree will require that the cache be updated to remain consistent. Invalidating a route cache entry represents the simplest case of the more general problem of "punching holes" in lower levels of the classifier so that packets can reach upper levels. Classifiers as a whole support this by only allowing updates to enter at the topmost level and then passing update information to the lower levels.

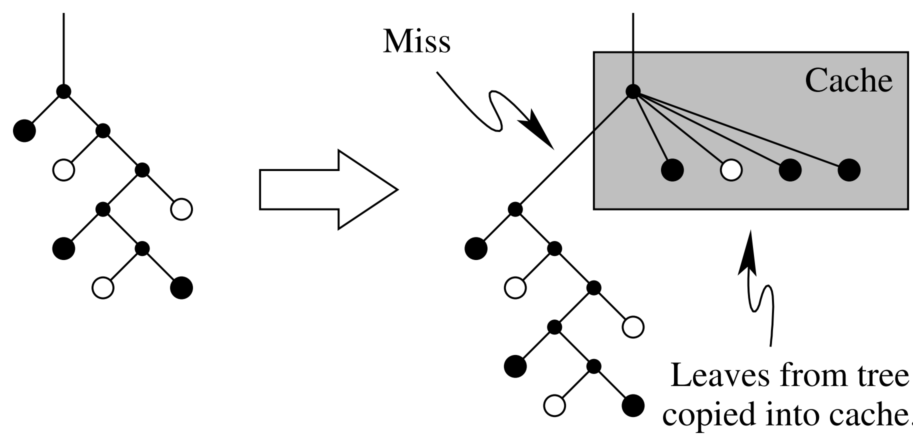

Figure 2.6: A partial classifier acting as a route cache. The left side shows a routing decision tree: internal nodes are decision points, leaves are path selections. The right side introduces a route cache. Misses must traverse the master tree.

2.2.3 Router API

This section outlines some of the router layer API function calls.

p* = createPath*(C, C_parms, F, F_parms, S, S_parms, Q_parms)

This function creates a new forwarding path, *p*, by instantiating a forwarder, *F*, and attaching it to the existing classifier, *C*, and scheduler, *S* through new queues. (Note that *C* and *S* implicitly identify the input and output ports, respectively.) Figure 2.7 illustrates this process with an example. *C_parms* include the demultiplexing key needed to identify the packets which should be redirected to this path. *F_parms* are passed to the forwarder, and include the processor reservation (cycles, memory) required by the forwarder. *S_parms* include the link reservation needed by the output scheduler. *Q_parms* are used to instantiate the queues that connect the components and include the switch reservation.

removePath*(p, parms)*

This function removes the existing forwarding path, *p*. The additional parameters indicate whether the path should be immediately terminated abandoning any packets in the queues or whether the path should be gracefully shut down by disconnecting it from the classifier first and then letting the packets drain out of the queues.

updateClassifier*(C, parms)*

This function allows updates (such as new routing tables) to be sent to a classifier.

2.2.4 Justification

While it is difficult to show that this is the "best" or "most correct" choice for a router abstraction, we will show that our abstraction is reasonable and effective. Clearly, this abstraction explicitly captures the essence of the routing task: classify, forward, and

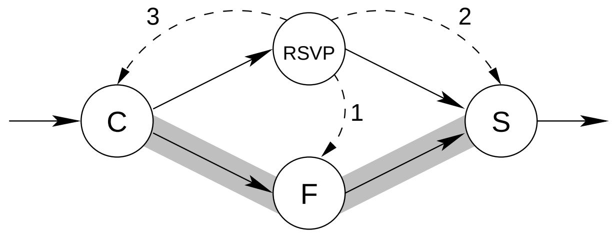

Figure 2.7: The basic operations performed by the **createPath** function. Here, an RSVP forwarder (1) instantiates a new forwarder, then (2) attaches to the appropriate output scheduler, and finally (3) attaches to the appropriate classifier.

schedule packets for output. Gottlieb and Peterson [20] have shown that these three elements model systems supporting a wide range of applications in their comparative study of extensible routers.

Based on the fact that the distribution of packet types is far from uniform (e.g., forwarded packets are far more common than routing protocol packets), we believe that a classification hierarchy is fundamental to all routers and not an artifact of VERA. We characterize the hierarchy by having (1) several, distinct stages and (2) simple stages near the bottom and complex stages near the top. The alternative to distinct stages is a single stage implemented as a lookup table. In general, the table would be indexed on all the bits of the header including any bits which might by IP options. This will yield a table so large that it could not be implemented. Therefore, any IP classification method will use a series of stages like those suggested in Section 2.2.2. To efficiently classify packets, it makes sense to order the stages to minimize the processing time based on the expected distribution of packet types. Simple, inexpensive tests that are likely to classify large numbers of packets are placed first (at the bottom); complex, expensive tests are placed last (at the top). The order in which to best place the stages defines the classification hierarchy.

Our router abstraction model dictates that every packet pass through a single classifier (that fully determines the forwarder and output scheduler), a single forwarder (with one input and one output), and a single output scheduler. Alternatively, one might use sequences of alternating classifiers, forwarders, and output schedulers. Such sequences, however, can always be modeled by a single, sufficiently sophisticated classifiers, a (potentially large) set of forwarders, and a sufficiently sophisticated output scheduler.

2.3 Discussion

Up to this point, we have described the essentially static router and hardware abstractions of VERA. That is, given a hardware configuration, we choose a static processor hierarchy and assign each port to a particular processor. Before introducing the dynamic aspects of thread decomposition, admission control, and function placement in Chapters 3 and 4, we argue that our architecture is both extensible and robust.

2.3.1 Forwarding Functions and Extensibility

In their comparative study of extensible routers, Gottlieb and Peterson [20] describe extensible routers as those including an element encapsulating forwarding functions that allow for arbitrary packet processing. Our architecture is extensible because it explicitly allows new forwarders to be instantiated on the router.

Note that VERA does not define a particular internal structure for forwarding functions due to the wide range of processor types (microengines to Pentiums) the architecture allows. Each processor is likely to have its own environment based on software engineering considerations. Examples of how forwarding functions might be implemented include:

Basic block / straight-line code: An implementation using straight-line, fixed-size code may be appropriate when the per-packet processing must be tightly bounded without the help of a pre-emptive scheduler.

C Functions: A forwarding function may be implemented as a simple C function.

Scout Paths / Click Modules: A forwarding function may be implemented using a sequence of C functions as in Scout [41], C++ modules in Click [34], or compositional elements as in Router Plugins [13].

Thread Sequences: A forwarding function may be multithreaded and be implemented across more than one processor. This may be especially appropriate when special purpose processors (e.g., encryption) are available within the router.

Regardless of how the forwarding functions are implemented it is because we explicitly provide a mechanism to install new forwarding functions that support new services, we conclude that VERA is extensible. In the next section, we argue that VERA is also robust.

2.3.2 Independent Impact and Robustness

Recall from the previous chapter that our definition of robustness consists of two parts: routers (1) must be able to read and classify packets at line speed and (2) must honor the processing guarantees they make. Because an architecture is not a router, we must modify the definition of robustness as it applies to an architecture as simply: a router architecture is robust if it admits a robust implementation. In the next two chapters, we will show that this is the case. In this section, we introduce *independent impact* as a key architectural attribute to support a robust implementation.

The placement algorithms described in Chapter 4 address the problem of assigning the forwarder code to the processors. Placing the forwarder code associated with a service on a particular processor or processors defines the forwarding path(s). As packets flow through the router, they will make an *impact* on the resources of the router. For example, they will consume processing cycles and switch bandwidth. A key design requirement for VERA is to create an environment where the packets comprising a flow have an impact on the resources of the router that is independent of the impact made by packets of other flows. We call this attribute *independent impact*.

We recognize that there will be *some* interaction between flows. For example, just having more entries in a task table can slightly increase latency. However, if the interaction is constant and small, we can model it as part of the overhead of the operating system. The main idea is that the impact be “additive” so that we can make the admission control / placement decision by adding the impact to the current router utilization and checking that the result does not exceed the capabilities of the router. By isolating flows and not exceeding the capabilities of the router, we ensure that the router remains robust. Independent impact is the method we use to ensure that the router can accurately track and allocate its resources.

For example, to submit a candidate service for installation on the router, the user (or the user’s agent) provides both a model of the impact on the router for an n-byte packet as well as the expected packet distribution and packet rate for the service. Where a forwarder can be instantiated in more than one way (e.g., on different processors or across multiple processors), the model provides the router impact for the various scenarios. By maintaining a table of committed resources within the router and comparing this against the independent impact of a candidate service, the placement decision can determine where (if anywhere) the code implementing the service can be placed.

The resource allocation and admission control techniques in Chapter 4 rely on the fact that the services have an independent impact on the system. In summary, we argue that our architecture is extensible by design and robust under the assumption that there exists a robust implementation. In the following chapters, we show that the implementation is extensible and robust.

Chapter 3

Implementation

This chapter describes the implementation of VERA. Section 3.1 describes our prototype hardware, Section 3.2 describes the Distributed Router Operating System and implementation techniques, and Section 3.3 concludes the chapter with an evaluation of how our prototype hardware implementation is extensible and robust.

3.1 Hardware

Figure 3.1 shows our router development testbed. It consists of a commodity motherboard connected to two different off-the-shelf line cards using a standard 33 MHz, 32 bit PCI bus. The motherboard is an Intel CA810E with a 133 MHz system bus, a 733 MHz Pentium III CPU, and populated with 128 Mbytes of main memory.

The first line card is a RAMiX PMC694 [52]. It contains a 266 MHz PowerPC processor, two 100 Mbps Ethernet ports, and 32 Mbytes of memory. The primary PCI bus (of the motherboard) is isolated from the secondary PCI bus (of the PMC694) with an Intel 21554 non-transparent PCI-to-PCI bridge. The secondary PCI bus is also 32 bits

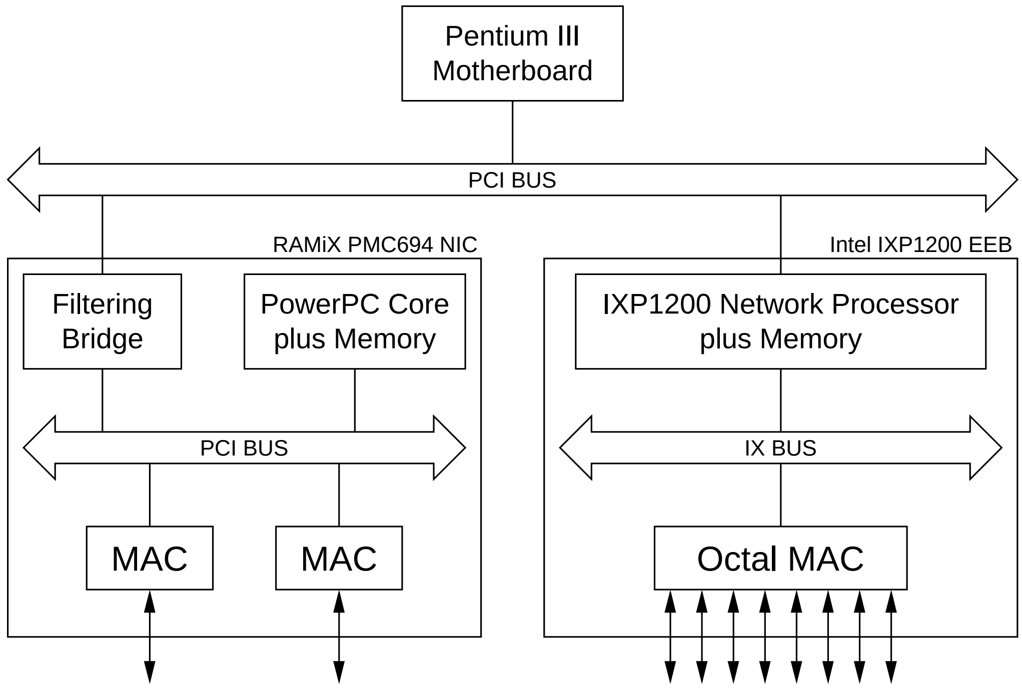

Figure 3.1: Testbed based on a Pentium III motherboard with both a PMC694 line card and an IXP1200 EEB line card.

and operates at 33 MHz. The PMC694 has a two-channel direct memory access (DMA) engine and queue management hardware registers used to support the Intelligent I/O (I_2O) standard [29].

The second line card is an Intel IXP1200 Ethernet Evaluation Board (EEB) [28]. A more detailed view of this line card is illustrated in Figure 3.2. The board contains a 200 MHz (5 ns cycle time)[1] IXP1200 network processor, a proprietary IX bus connected to eight 100 Mbps Ethernet ports[2], 32 Mbytes of SDRAM, and 2 Mbytes of SRAM. Like the PMC694, this board also has a two-channel DMA engine and I_2O support registers.

[1]Actual speed is 199.066 MHz.
[2]The IXP1200 EEB also has two 1 Gbps (fiber) Ethernet ports which we do not use.

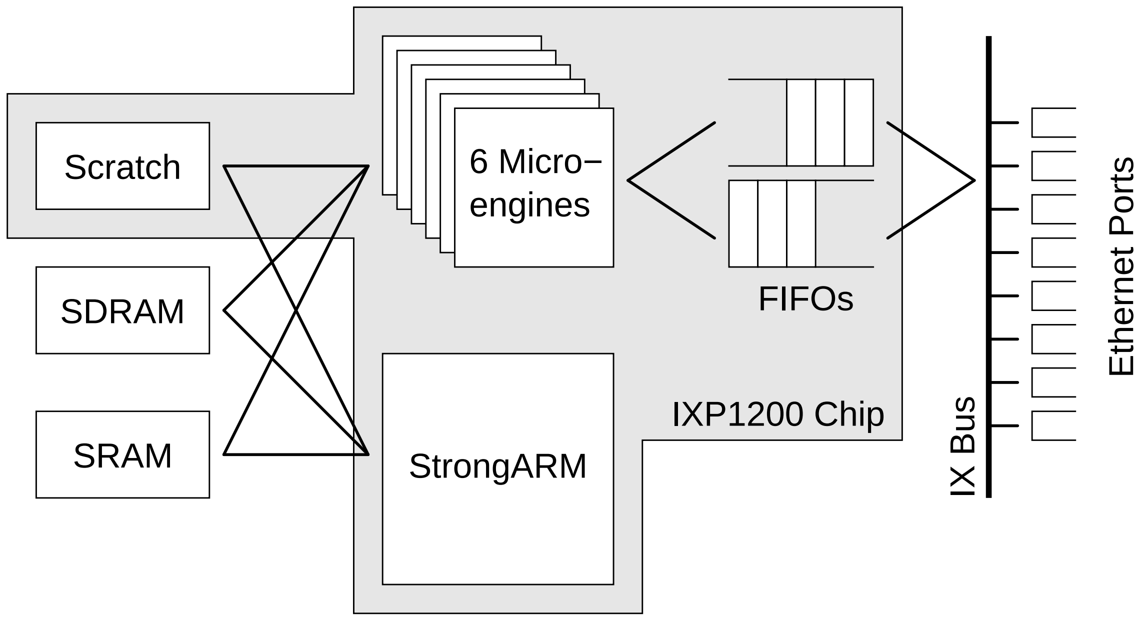

Figure 3.2: Block Diagram of an IXP1200 EEB. The shaded area is the IXP1200 chip. The PCI interface (not shown) connects to the StrongARM.

IXP1200 Network Processor

The IXP1200 chip contains a general-purpose StrongARM processor core and six special-purpose microengine cores all running at 200 MHz. The chip contains separate interfaces to SDRAM and SRAM. In addition, there is a 4 Kbyte on-chip scratch memory. The StrongARM has a 4 Kbyte instruction cache and an 8 Kbyte data cache. Each of the six microengines supports four hardware contexts for a total of 24 contexts. Not shown in the figure is a 4 Kbyte instruction store associated with each microengine. The StrongARM is responsible for loading these microengine instruction stores.

The chip also has a pair of FIFOs used to transfer packets to and from the network ports across the IX bus. These are not true FIFOs in the sense that each has a single input, a single output, and no address lines; rather, each "FIFO" is an addressable 16 slot $\times$ 64 byte register file. It is up to the programmer to use these register files so that they behave as FIFOs. Data is transferred across the IX bus in 64-byte chucks of data called *message packets* (MPs). The FIFO is sized to hold 16 MPs. Ethernet frames that are larger than 64 bytes are broken into smaller MPs by the MACs and transferred sequentially.

Although not explicitly prescribed by the architecture, the most natural use of the SDRAM is to buffer packets. This is not only a function of size (256 Mbyte address space), but also of speed. The SDRAM is connected to the processor by a 64-bit, 100 MHz data path, giving a peak bandwidth of 6.4 Gbps. This is sufficient to support the $2 \times 8 \times 100$ Mbps $= 1.6$ Gbps total send/receive bandwidth of the 100 Mbps network ports. Similarly, SRAM is a natural place to store the routing table, along with any necessary per-flow state. The SRAM is connected to the processor by a 32-bit, 100 MHz data path, giving a peak bandwidth 3.2 Gbps. (We also note that the 4 Gbps peak bandwidth of the 64-bit, 66 MHz IX bus is sufficient to support the 1.6 Gbps bandwidth of the 100 Mbps network ports.)

3.2 Distributed Router Operating System

The distributed router operating system (DROS) is a software layer that runs on each processor in the router. For high-speed, general purpose processors (such as a Pentium on a PCI motherboard), this may be a layer on top of a general purpose OS such as Linux. For processors on line cards, this layer may represent the entire operating environment (i.e., the DROS layer may run directly on the hardware). As stated in Chapter 1, the purpose of the DROS is to provide an execution environment for the forwarding functions and bridge the semantic gap between the high-level router abstraction and the low-level HAL; the OS *implements* the **createPath** call using the data movement and hardware queue support functions of the HAL. In addition to tying together these core abstractions, the OS provides a computation abstraction in the form of a thread API and a memory abstraction in the form of both a buffer management API and an internal routing header datatype. As mentioned in Chapter 2, a key design goal for VERA is to partition the router's resources so that services have an independent impact on the system. In the following subsections, we outline the major features and abstractions provided by the operating system.

3.2.1 Processor Hierarchy Revisited

Recall that the processors are organized into a hierarchy. At router initialization time, administrative message queues are created from the master processor to each of its child processors, and so on down the hierarchy. The single, master processor maintains the master copies of the routing tables and controls the overall operation of the router. Since each processor operates independently and has its own thread scheduler, the control the master processor has over the other processors is, by necessity, coarse grained.

The processor hierarchy nicely maps to the classification hierarchy of Figure 2.5. The first partial classifier, C_1, always runs on the processor managing a particular input port. The last partial classifier, C_n, always runs on the master processor. Intermediate classification stages can be mapped to either the current processor, or the next processor up the processor hierarchy. Each classifier exports a single interface. This interface is exported only on the master processor. Updates to the classifier are made via this interface and then trickle down the classification hierarchy as needed. This means that the OS must propagate router abstraction level calls to **updateClassifier** through the processor hierarchy to the master processor where they can be fed into the classifier at the top level.

3.2.2 Router Primitive Decomposition and Scheduling

The need to quickly classify packets is at odds with our desire to be able to look arbitrarily deep into packets. The compromise VERA makes is to use a per-packet processor-cycle classification budget. The minimum number of cycles in the budget depends on the line speed of the queues feeding the processor as well as the processor cycle speed and is set by the router architect. Within this fixed budget, each packet must be classified enough to determine its service quality. That is, the router determines which service to charge on behalf of the packet.

A Case for Separate Threads

Within our router, there is a fundamental tension between (1) supporting arbitrary classification and forwarding functions, and (2) supporting QoS. Qie, *et al.* [51] shows that to support QoS effectively on a uniprocessor software router, one should use separately scheduled threads for classification, forwarding, and scheduling. In fact, to support QoS, it is important that classifiers be able to determine the fate of packets at line speed. If this were not the case, high-priority packets could be delayed or missed while the system classifies other incoming packets that are eventually determined to be of a low priority. QoS support is related to robustness. A router that is not robust will not always be able to uphold its service guarantees and therefore cannot truly support QoS under all circumstances.

Our design stipulates that output schedulers not modify packets. This allows the scheduler to make its thread scheduling decision based on the state of the queues going into and coming from the output scheduler without needing to estimate the amount of processing which might need to be performed on the packet. Because no packet processing occurs in the classifier or output scheduler, all processing must occur in the forwarder. We have chosen not to perform link-layer processing in the output scheduler, the forwarder can perform arbitrary processing at the link layer. The downside is that if there are N different media types in the router, each processor must be capable of handling all N link layers. However, we expect N to be small enough (in fact, usually 1) that $N \times N$ translation will not be an issue.

Thread Scheduling Classes

We divide threads into two classes: statically scheduled and dynamically scheduled. *Statically scheduled* threads are not charged to a particular flow or service and use a fixed share of the processor and run on a fixed schedule that is determined *a priori*. For example, classification (to the point of identifying the QoS) is statically scheduled. *Dynamically scheduled* threads are charged to a particular flow or service and use a share of the processor based on a reservation as well as the current state of the router (e.g., queue depth).

Specifically, a statically scheduled processor guarantees that it will be able to process packets arriving at line speed. The consequence is that a statically scheduled processor knows exactly how many cycles are available for each packet, and admits only functions whose worst-case behavior fits within this cycle budget. Should packets arrive at a lesser rate, or some fraction of the per-packet cycle budget be unallocated, the excess cycles are simply wasted. For example, we choose to statically schedule the IXP1200 microengines because they must be able to receive-and-forward packets at the aggregate speed of the connected ports [55].

In contrast, a dynamically scheduled processor accommodates the possibility that packets may arrive at a far greater rate than it can process them, either because it admits functions with large processing costs relative to the worst-case rate at which packets can arrive, or because it admits functions with variable processing costs. In this case, packets must be segregated according to the treatment they are to receive (i.e., the function that is to be applied to them), with each function given a share of the processor's cycles. Such processors still require an admission control decision—for example, to ensure that the average cycle demand of the admitted functions does not exceed the processor's capacity—but the dynamic scheduler is able to allocate cycles to different functions based on the actual workload (packet arrival rate) it is experiencing. A proportional share scheduler is a likely implementation since it guarantees that the function (flow) receives at least the cycle rate it requested, and fairly allocates any unused capacity among the active functions [51].

After new forwarder threads are instantiated, they must be scheduled along with all the other classifier, forwarder, and output scheduler threads. Because our architecture supports a heterogeneous distributed processing model, the OS must provide support for scheduling computations across the entire router. In Chapter 2 we defined a *scheduling domain* as a tightly bound set of processors (usually, a single processor) running a single thread scheduler. Because the amount of processing required on each packet is small, we have chosen not to have a single, fine-grained scheduler for all the processors of the router. Instead, each processor runs a completely independent scheduler. The master scheduler (running on the master processor) provides only coarse grain adjustments to the schedules for each scheduling domain to guide their independent scheduling decisions.

Section 3.2.9 describes a statically scheduled environment for the microengines of the IXP1200 network processor. The input threads have enough extra processing cycles in their budget to perform non-trivial packet modifications. In this case, we can combine classification and forwarding into a single thread.

3.2.3 Internal Packet Routing

It is well known that routers should internally copy data as little as possible. Our architecture helps reduce the amount of data copying by sending an *internal routing header* (IRH) (rather than the entire packet) from processor to processor. This internal routing header contains the initial bytes of the packet plus an ordered scatter/gather list of pointers to blocks containing the data of the packet. A reference count is associated with each block; the OS uses this count to determine when a block can be recycled. (The classifier changes the reference count from its nominal value of one when a multicast or broadcast packet is detected; the reference count is set to the number of destinations.)

An interesting property of IP routers is that in most cases only the header need be modified by the forwarder; the body/payload of the packet remains unchanged. When activated by the thread scheduler, a forwarder first reads the next internal routing header from its input queue, fetches any (remote) packet data it needs and then performs the computation. After processing the packet, the forwarder sends the updated internal routing header to its output queue (which is connected to the appropriate output scheduler). It is the responsibility of the queue API to make sure all the packet data is local to the output scheduler before the internal routing header is placed on the queue. Because the classification hierarchy and the forwarder on the invoked path may not have needed the entire packet to complete the classification and requisite forwarding function, the packet may be scattered in several blocks across several processors. Anytime a thread moves a block, the thread must also update the internal routing header to reflect the new block location. When the internal routing header reaches the output scheduler's processor, the data must copied to local memory before the internal routing header can be placed on the output scheduler's inbound queue. Figure 3.3 illustrates the sequence.

1. The forwarder, F, sends an IRH to the output scheduler, S.

2. The *queue server (QS)*, preprocesses the IRH to determine the location of the packet data.

3. The QS uses the HAL data movement primitives to fetch the packet data.

4. The QS modifies the IRH to indicate that the data is now local and places it on the input queue where it is visible to S.

5. S reads the IRH from the queue.

6. S directs the data from local memory to the device queue.

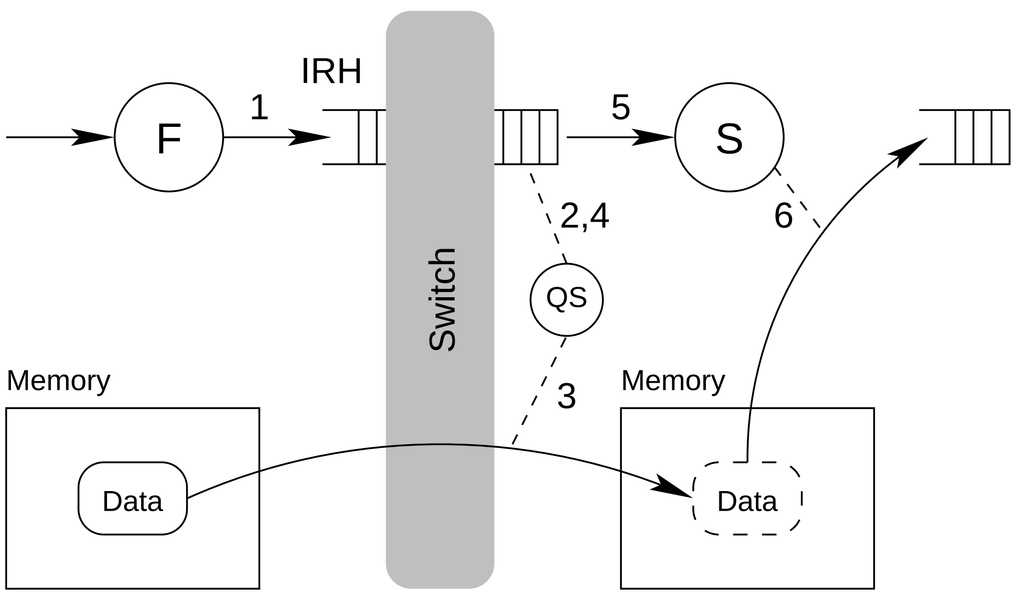

Figure 3.3: This shows the steps performed by the OS when an internal routing header (IRH) is sent from a forwarder, F, on one processor to an output scheduler, S, on another processor. (See text.)

Until now we have discussed the issues with moving the data through the router. Eventually, these buffers must be recycled when the packet data is no longer needed. The

HAL data movement commands are really data *copying* commands. It is the responsibility of the OS to manage the buffers. When we wish to *move* a buffer referenced by an IRH, we send a release message to the source processor. The processor decrements the reference count associated with the block(s) and reuses any whose count has reached zero.

3.2.4 Distributed Queues

Queues are the mechanism for moving packets from thread to thread. In general, the threads are on different processors, and thus, the queue will span a switch. As defined in Chapter 2, these are called *distributed queues*. Because every packet will pass through at least one queue on its path through the router, we must take special care to ensure efficient implementations on the target hardware. In implementing distributed queues we consider the following aspects:

Data Movement: This basically means to move packet references through queues rather than the actual packet data. For non-distributed queues (i.e., between threads on the same processor), this is the natural implementation. For distributed queues, the overhead involved in copying or sending data from one processor to another is high enough that it often makes sense to go ahead and send some of the data. (This is part of the rationale behind the IRH.) Another consideration is that the pointers within the IRH must have meaning to the processor that uses them.

Data Structure Overhead: Because the router must support thousands of flows, the OS must efficiently implement queues and, because many flows will be dynamic, efficiently create and destroy queues.

Hardware Support: In some cases (including our prototype system), the underlying hardware has capabilities we can leverage to implement queues directly. Unless the hardware can directly support thousands of queues, VERA uses the hardware queue support to multiplex a larger number of virtual queues on top of the hardware queues. When a virtual queue is established, it is given an identification key. Each queue element is tagged with the key of the destination virtual queue. A demultiplexing step occurs on the receiving end to assign the incoming item to the correct queue (cf. Figure 3.5 on page 58).

With a shared memory system such as the PCI bus, we have several implementation options available. These are discussed below. To support a large and varying number of queues efficiently, we will settle on Method 4. Figure 3.4 gives a visual interpretation of the following methods:

Method 1: Separate queue from each processor for each flow. The problem with this approach is that it requires *many* queues. Any efficient implementation will require that the queues use a fixed-sized, pre-allocated block of memory. A large number of queues can lead to inefficient use of memory. In addition, having the scheduler check a large number of queues will introduce additional processor overhead.

Method 2: Separate queue from each processor for each QoS level. By pushing the multiplexing of the flows at a given QoS level back to the sending processor, we can reduce the number of input queues to one per processor for each QoS level. This leaves us with the problem that the output scheduler should choose packets from the set of queues for a given QoS level on a first-come first-served basis. (Note that this is also a problem with the previous method.) To support this choice requires that the sending processor attach a global timestamp to each packet. Supporting a global timestamp would introduce potentially unacceptable overhead. In addition, it is not clear that multiplexing flows at the same QoS level would sufficiently reduce the number of input queues.

Method 3: Separate queue for each QoS level. By using a separate queue for each QoS level and letting the sending processors contend for the queues, we eliminate the need for a global timestamp. The problem we introduce is that the processors must contend for a semaphore for each queue to be able to insert an item. Without special hardware support, implementing a multiple writer queue over the a would require the processors to acquire and release semaphores using software techniques (for example, spinning on remote memory locations [21]). As in Method 2, it is not clear that multiplexing flows at the same QoS level would sufficiently reduce the number of input queues.

Method 4: Single queue. This method has the advantage that global timestamps are not needed and there is a single input queue independent of the number of QoS levels. As in Method 3, processors must contend for access to the queue.

By electing Method 4, each processor uses a single input queue and each packet that is moved from one processor to another must be re-examined to some extent by the receiving processor's queue server thread (acting as a classifier). In addition, the sending processor will need to make a scheduling decision to determine when the packet can be moved across the switch. In effect, each individual processor acts as a mini-router: classifying packets from each of its attached switches, applying some (possibly null) function to these packets, and scheduling the packets for output. As depicted in Figure 3.5, this happens on every switch that a packet traverses. Generally, only one of the processors hosts the forwarding function. The other processors use a null (pass-through)

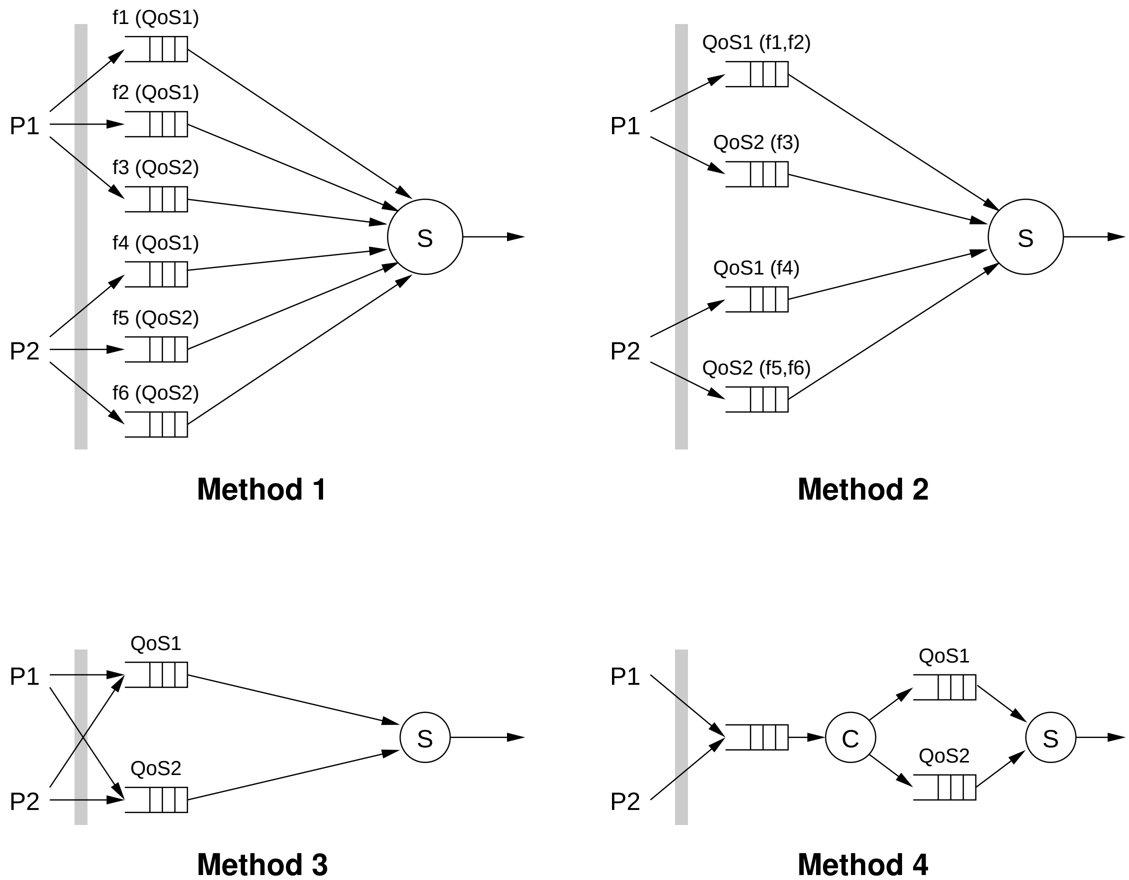

Figure 3.4: Distributed Queue Implementation Methods. P1 and P2 indicate processors which are on the opposite side of the switch (gray vertical line). Flows are indicated by fn. Quality-of-service levels are indicated by QoSn. S indicates the output scheduler. C indicates a classifier/demultiplexer.

forwarder. Note that the internal mux/demux stages as well as the null forwarders are hidden from the application developer. While it may appear that we are simply pushing the Internet routing problem onto the processors in our router, this is not the case. Unlike the Internet, we have a central authority (the master processor) that has global knowledge of the routing table and it pushes pre-processed forwarding tables to each processor.

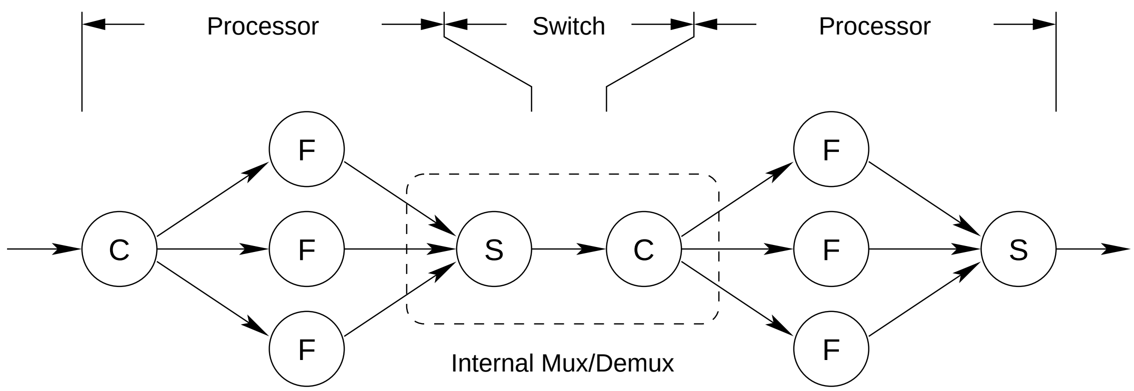

Figure 3.5: Nested Routers. In practice, each processor in the router must classify, forward, and schedule packets. (The internal mux/demux is hidden from the application developer.)

3.2.5 Indirect Bandwidth Management

Given finite memory, all routers will have at least one choke point (i.e., potential bottleneck). For example, if all the packets arriving on two ports are routed to a third port (whose bandwidth is less than the sum of the two ports), then some packets must eventually be dropped at or before the point where the packets merge. Buffering will help to avoid packet loss if the average rates match. This is the foundation of statistical multiplexing. Preventing choke points on shared resources spanning scheduling domains from becoming bottlenecks represents an interesting design challenge. Figure 3.6 depicts a four-port router using a shared switch with two established packet flows. Note that the switching paths for the two flows contain a common switch but *do not* contain a common processor. Without some form of coordination between the line cards, the best effort (BE) flow could use so much of the switch bandwidth that it prevents the QoS flow from meeting its guarantees. (In the case of a PCI bus, the bus arbiter uses a fair algorithm to decide which of a set of requesting bus masters will own the bus.) To support both BE and

QoS flows, VERA superimposes an *indirect bandwidth management* technique on top of shared, decentralized switches. This allows the bus to be shared in a non-fair way (e.g., giving a QoS flow priority over a BE flow). Carefully managing the internal switching resource and preventing flows from interfering with one another is an important part of maintaining the property of independent impact which, in turn, is an important part of robustness (cf. Section 2.3.2).

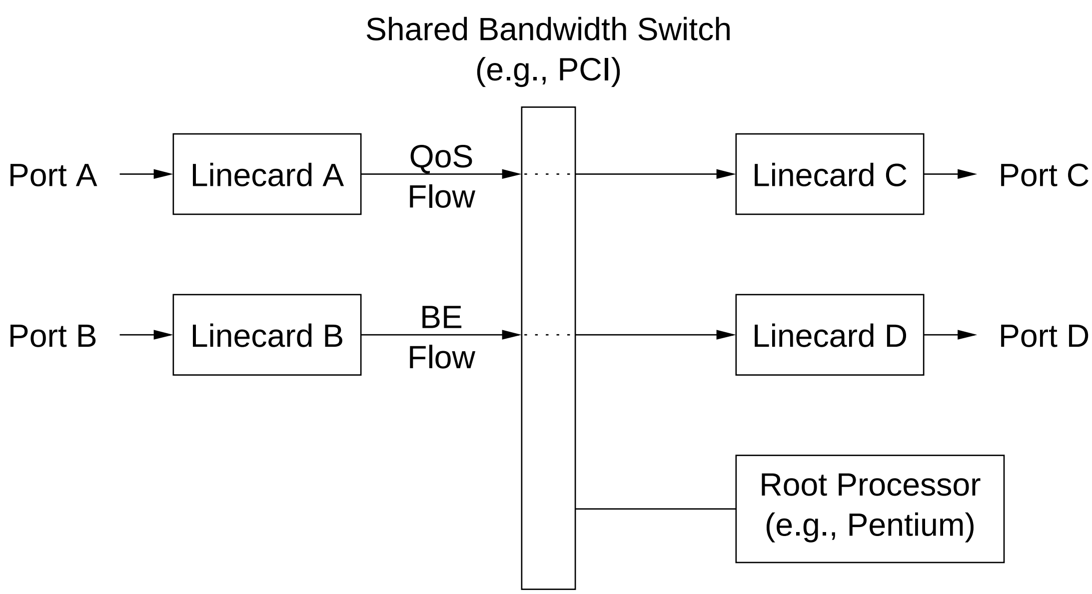

Figure 3.6: QoS / Best-Effort Interference on a Shared Switch. A QoS flow between two ports can be adversely affected by a best-effort (BE) flow between two other ports on the same shared switch. A throttling mechanism coordinated by the root processor gives each line card a share of the switch resource.

Two methods of indirectly managing the bandwidth include token passing among the bus masters and a centralized controller on the root processor. In either case, the management only applies to packet traffic. The underlying hardware allows any processor to arbitrate for the resource at any time to send administrative messages. This overhead traffic should be accounted for when determining the budget allocated to the following techniques:

Token Passing: This technique passes a token from processor to processor. When a processor holds a token, it is entitled to use the shared switch for some amount of time before passing the token on to the next processor in the cycle.

Periodic Reporting: With this technique, depicted in Figure 3.7, each processor periodically reports the state of its queues to the root processor. The root processor uses this information to make a global decision and periodically send allowances to each processor on the switch.

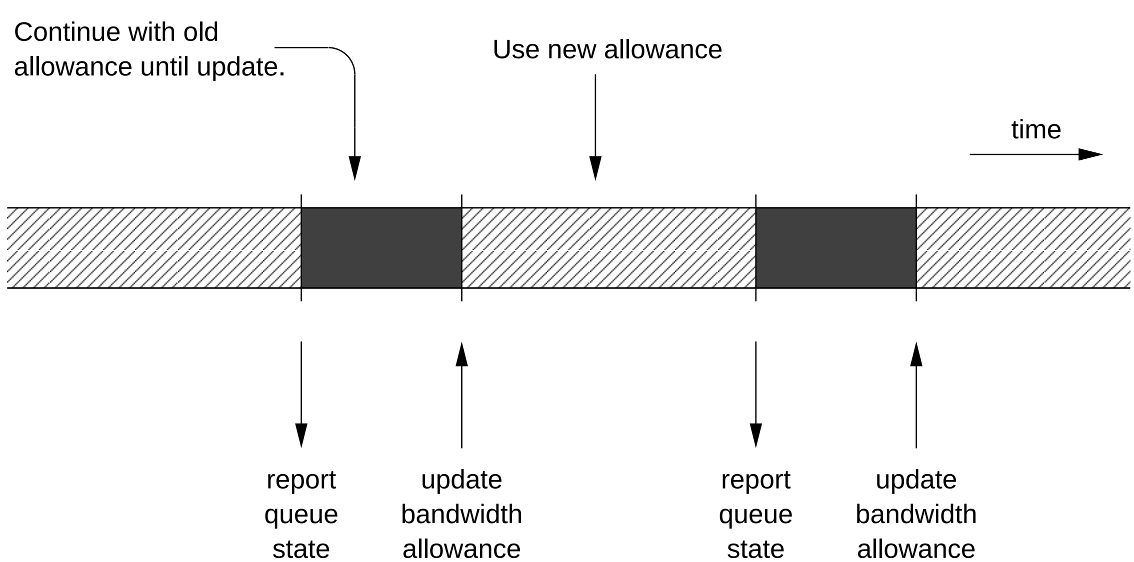

Figure 3.7: Periodic reporting of queue states and bandwidth allowance distribution.

Of these two techniques, VERA uses periodic reporting. Because the underlying hardware already arbitrates for the use of the switch, we want to avoid the additional overhead incurred by the arbitration that token passing generates. Also, if a processor holds a token but only has BE flows waiting to use the switch, it does not have enough information to determine how much (if any) of the switch resource it may use. This information could be passed with the token but it would require that every processor implement an algorithm to determine appropriate switch usage and each processor would have a different picture of the state of the queues depending on their position in the token ring.

3.2.6 PCI Switch Implementation

The common denominator of all IP routers is that they move data from an input network interface to an output network interface. The HAL exports an API that allows any of the processors in the router to either push (put) or pull (get) data to or from any of the other processors in the router.

We note that the PCI bus efficiency is highest when data is transferred in long *bursts*. This is a direct consequence of the fact that a PCI bus transaction consists of an *address phase* followed by one or more *data phases*. The address phase specifies the address of the transfer for the first data phase; each subsequent data phase within the same bus transaction occurs at the next higher address. The address phase takes one PCI clock cycle; in the best case, each data phase takes one PCI clock cycle. For each data phase, either the master or the target may insert *wait states* (each of which takes one PCI clock cycle). Note that read operations have a mandatory wait state in their first data phase and each transaction must be separated by at least one PCI clock cycle. By transferring data in bursts (bus transactions with many data phases), the overhead (per byte transferred) of the address phase is reduced.

Because processors cannot consistently generate these efficient bursts, they are often augmented with DMA engines specifically designed to generate bursts to transfer blocks of data. Both the PMC694 and the IXP1200 have DMA engines that consistently generate burst transfers. Note that our Pentium motherboard does not have a DMA engine for the PCI bus and therefore must implement data movement in software—referred to as *programmed I/O* (PIO). An additional advantage of using DMA over programmed I/O is concurrency; after issuing a DMA request, a thread can either move on to other computations or yield to another thread.

Due to the freedom in the PCI specification, different board designs will exhibit different performance characteristics. Both the host processor (the Pentium III) and the on-board processor of the line cards can arbitrate for and then become the PCI bus master which allows them to initiate transfers. By using read or write instructions (or a DMA engine if available) a processor can pull or push data across the bus. Our experimental results are discussed in Section 3.3.

Managing the DMA Engine / PCI Bus Resource

Since there is only one DMA engine per line card, the DMA engine becomes a critical resource that is explicitly managed by the HAL. Moreover, because the PCI bus is a shared resource and there is no inherent bandwidth reservation mechanism, the HAL must coordinate among the processors when using its DMA engine. Thus, rather than simply hide the DMA engine beneath a procedural interface, we dedicate a server thread to the DMA engine on each processor. This thread coordinates with its peers to allocate the shared bandwidth available on the bus, as well as manages the local engine and DMA queues. The server supports two DMA work queues: a low-priority queue for packet traffic and a high-priority queue inter-processor control messages (e.g., messages that release packet data buffers.)

It would not be unreasonable for us to use a procedural interface to the DMA engine which would potentially yield and then let the local scheduler include the state of the DMA controller in the overall scheduling decision. We choose not to use this approach for two primary reasons. First, the DMA engine operates asynchronously to the processor; the DMA engine can even interrupt the processor when it has finished. Second, the DMA engine was "factored-out" of the local scheduler as a software engineering decision; by decoupling the DMA engine from the local scheduler, the design and implementation of each becomes more modular and less complex. The potential inefficiencies introduced by this decision can be explored as part of future work.

Because our hardware model allows multiple intervening processors and switching elements between any two processors, the intervening processors must store-and-forward packets. While this appears to be the problem that the router as a whole is trying to solve (IP routing), the problem is significantly easier because, unlike the Internet, we have a centralized controller in the master processor. The HAL hides the details of any necessary forwarding.

Distributed Queues over PCI

As discussed in the previous section, we elect to implement our distributed queues using a single input queue followed by a classifier that quickly separates the stream into separate QoS queues (Method 4). By using a single queue, the sending processors must contend for a semaphore to be able to insert an item on to the queue. Because the current version of the PCI bus specification [46] does not support bus locking by arbitrary bus masters,[3]

[3]Interestingly, prior versions of the PCI specification did support bus locking.

we either need special hardware support or expensive software techniques to implement multiple write queues.

Fortunately, the line cards we selected support I_2O and have hardware registers that directly implement multiple reader or multiple writer queues. They do this by hiding a FIFO behind a single PCI mapped register. Each element in the FIFO is a 32-bit (pointer-sized) integer. When a processor reads the hardware register, there is a side effect of updating the FIFO pointer. The read and update happen atomically. Because these components only have support for two FIFOs in each direction, we can only support a single queue.

By using I_2O support, we are manipulating *pointers* to IRHs. In order to effect the transfer of an IRH, we use two hardware-level FIFOs to implement a distributed queue. One FIFO contains pointers to empty IRH frames, and one FIFO contains pointers to to-be-processed IRH frames. Putting an IRH onto a queue involves first pulling a pointer to a free frame from the free-frame FIFO, filling the frame using the data movement primitives in the HAL, and then placing the pointer on the to-be-processed FIFO. Getting an IRH from a queue involves first retrieving a pointer from the to-be-processed FIFO, processing the data, and then returning the block to the free pool by placing its pointer on the free-block FIFO.

PCI Low-Level Hardware Abstraction

This section outlines some of the PCI low-level hardware layer API function calls. The following two functions hide details of how the hardware moves data:

putData*(local, remote, size)*

This function pushes a block of data of length *size* from a *local* address to a *remote* address using DMA or PIO, whichever is fastest.

getData*(remote, local, size)*

This function pulls a block of data of length *size* from a *local* address to a *remote* address using DMA or PIO, whichever is fastest.

The following three functions hide details of the hardware FIFOs:

f = **allocFIFO***(dir, depth)*

This function allocates a hardware FIFO, *f*, from a fixed-size pool of FIFOs. The FIFO is configured to hold *depth* entries each of which is a pointer-sized integer. The direction parameter, *dir*, can be set to either **`incoming`** or **`outgoing`**. An incoming FIFO supports multiple remote writers and an outgoing FIFO supports multiple remote readers. Note that the local processor (which made the call to **allocFIFO**) cannot write to incoming FIFOs and cannot read from outgoing FIFOs. This restriction along with the choice of pointer-sized items is made to allow efficient implementations on systems that support I_2O.

insert*(fifo, item)*

This function inserts the *item* (a pointer-sized integer) on the given *fifo*. The *fifo* must be either a locally allocated outgoing FIFO or a remotely allocated incoming FIFO.

item = **remove***(fifo)*

This function removes the *item* (a pointer-sized integer) from the given *fifo*. The *fifo* must be either a locally allocated incoming FIFO or a remotely allocated outgoing FIFO.

Because we take advantage of I_2O support as the basis for **insert** and **remove**, we also must live with the restrictions of I_2O. Specifically, the local processor cannot access the registers used by the remote processors and vice versa. This is the reasoning behind the corresponding restrictions in **insert** and **remove**. The benefit is that the implementation of **insert** or **remove** can be as simple as a write or read of a memory-mapped register.

Other Switching Hardware

Our implementation effort up to this point has focused on the PCI bus as a switching element; however, we considered other technologies when defining the HAL as well. There are two primary considerations when using other switches. First, the HAL defines operations for both pushing and pulling data across the switch. This is a natural match for a bus, which supports both read and write operations, and is consistent with interfaces like VIA [9] (which form the basis for interfaces like InfiniBand [26]). However, on a switch component that supports only push (send) operations (e.g., an Ethernet switch), the pull operation will have to be implemented by pushing a request to the data source, which then pushes back the requested data. Second, the bus offers only best-effort, shared bandwidth. On a switching element that supports either dedicated point-to-point bandwidth (e.g., a crossbar) or virtual point-to-point bandwidth (e.g., InfiniBand channels), the role of the DMA server thread diminishes. In effect, the hardware supports the point-to-point behavior that the HAL requires.

Another possibility is to connect the line card directly to the memory bus of the processor. This approach was used in the line card of the SHRIMP–II system [5]. A difficulty with this approach is that its method of sending packets without explicit commands from the processor (called *automatic update*) requires custom hardware to make direct performance measurements [33, 40]. Another difficulty is that commercially available PC hardware generally does not supply a direct connection to the memory bus. This can be circumvented by using a dual-processor motherboard and using a custom-designed line card in place of one of the processors [22].

3.2.7 Prototype Implementation

For our prototype, we chose Linux for both the development environment as well as the implementation operating system for the Pentium. We discuss the development environments and then the Pentium environment here and then continue with a discussion of the line card firmware and microengine environment in Section 3.2.8 and Section 3.2.9, respectively.

Development Environment

We use the GNU C tools for the Pentium, StrongARM, and PowerPC processors. All run on the Pentium under Linux. The Pentium compiler is native while the StrongARM and PowerPC compilers are cross-compilers. We use the Intel-supplied microcode assembler to assemble IXP1200 microcode. We run the microcode assembler on the WINE Windows emulation environment on Linux. The toolchain to create an executable image for the IXP1200 is shown in Figure 3.8. The PMC694 toolchain is simpler as the PMC694 does not contain microengines.

From an application developer's point of view, the prototype VERA environment is a runtime system. In this context, the DROS is an "application." The DROS, in turn, provides execution environments for router extensions. The directory structure of the runtime system in shown in Figure 3.9. The runtime system contains over 40,000 text lines of code. A template for the directory structure of an application is shown in Figure 3.10. The directory trees use the following naming conventions:

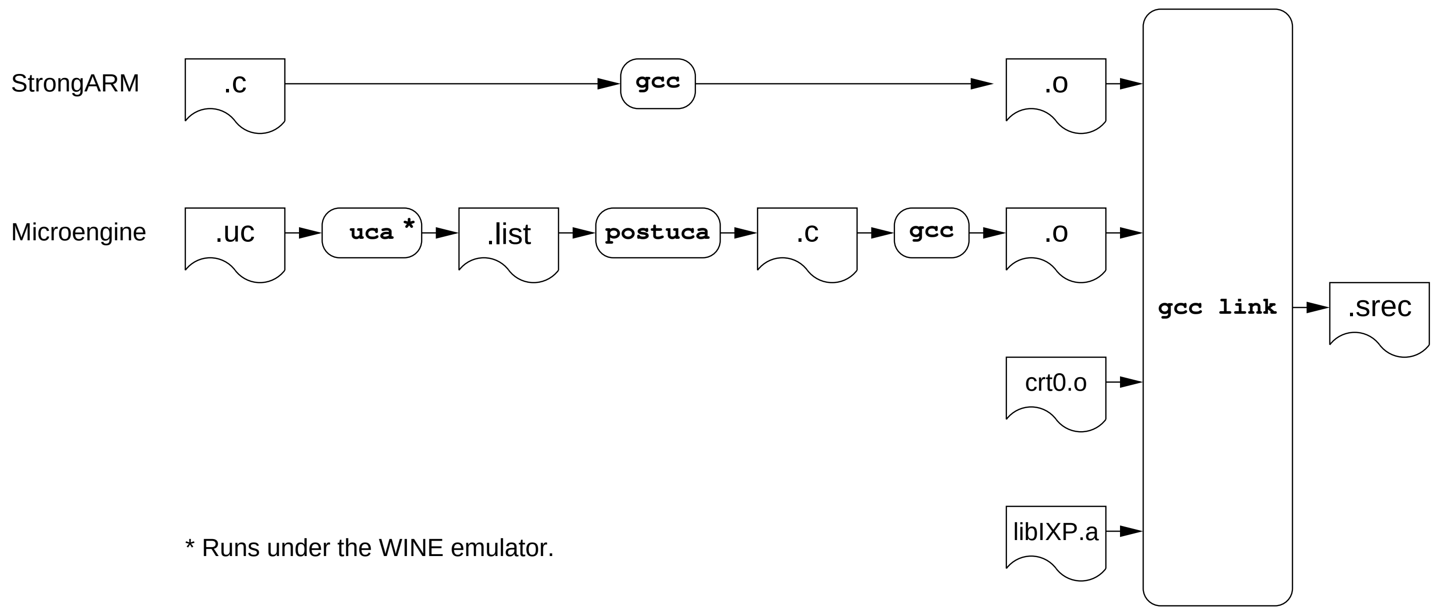

Figure 3.8: VERA's IXP1200 Toolchain

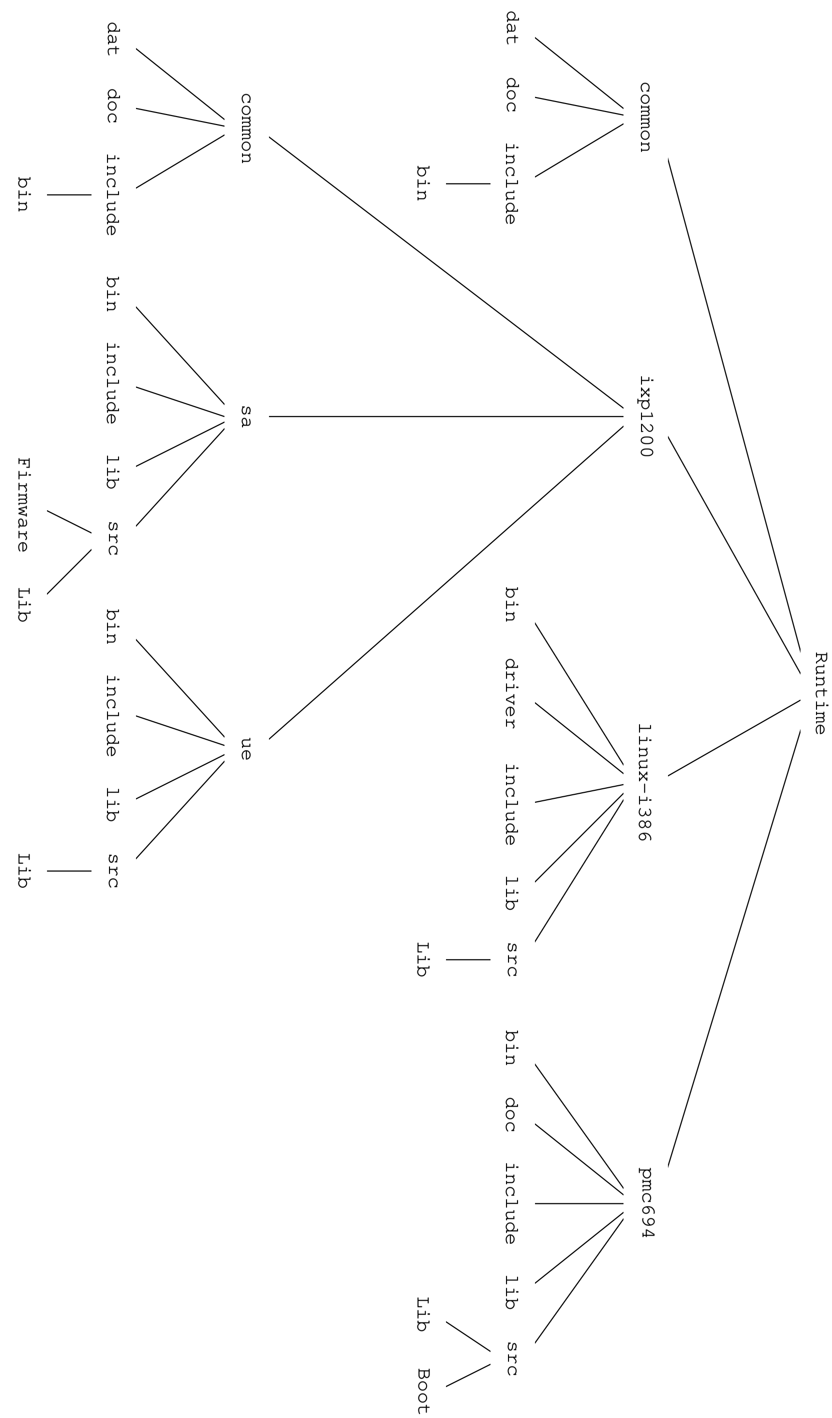

Figure 3.9: Runtime directory structure supporting both the IXP1200 EEB and PMC694 line cards under Linux running on a Pentium.

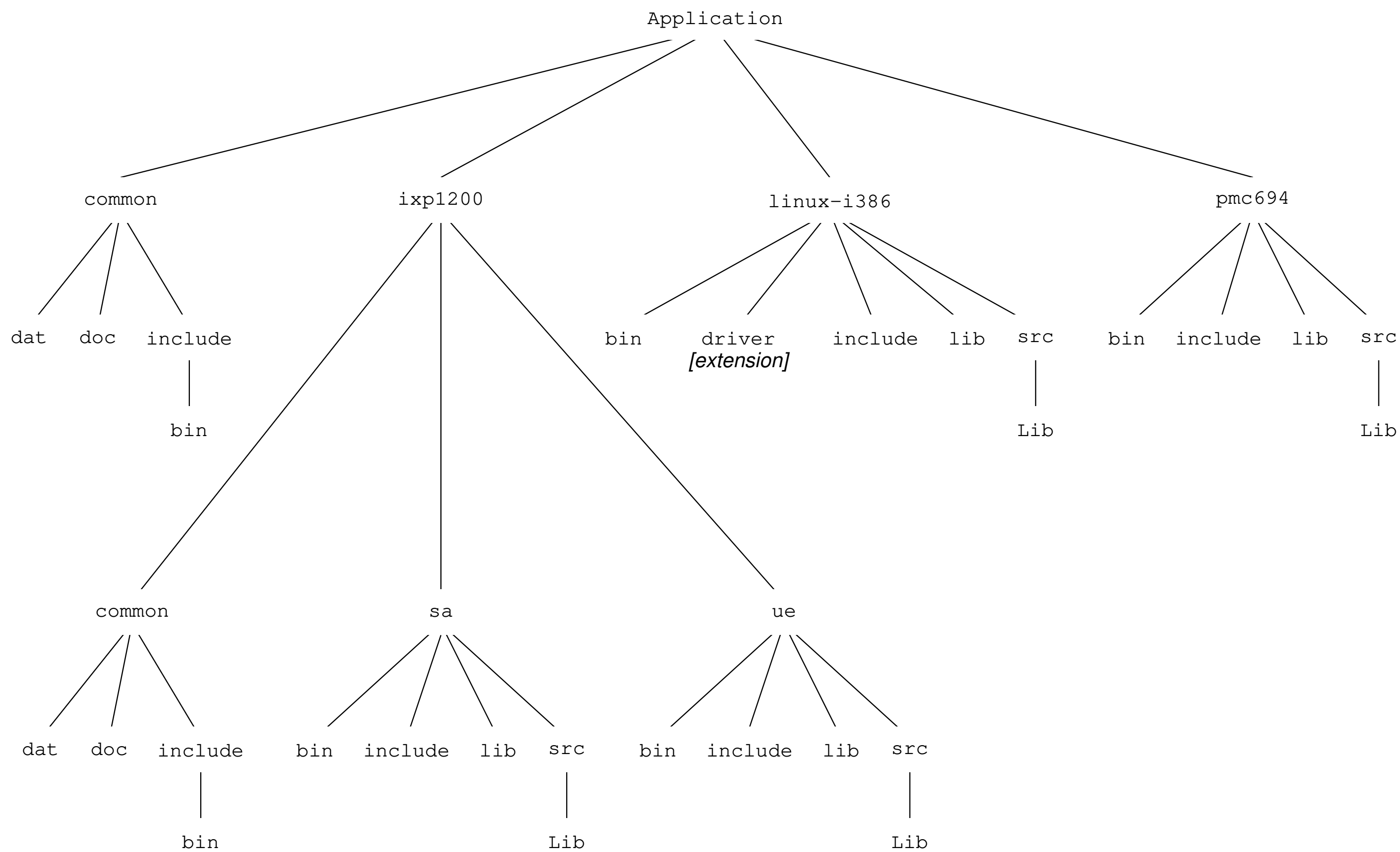

Figure 3.10: Application directory structure template supporting both the IXP1200 EEB and PMC694 line cards under Linux running on a Pentium.

`linux-i386`

The Linux/Pentium directory tree.

`linux-i386/driver`

The Linux device driver subdirectory. In the `Runtime` tree, this is the `vera.o` driver module. For the `Application` tree, this contains the code for the driver extension modules.

`ixp1200`, `pmc694`

The IXP1200 EEB and PMC694 subdirectory trees.

`ixp1200/sa`, `ixp1200/ue`

The StrongARM and microengine subdirectory trees for the IXP1200.

`common`

A `common` subdirectory contains files that are shared by sibling directories. For example, `ixp1200/common` contains code common to `ixp1200/sa` and `ixp1200/ue`.

`include`

An `include` subdirectory contains header files which describe libraries in `../lib`. They may have a `bin` subdirectory containing compiled executable code in the form of a header file.

`src`

A `src` subdirectory contains code that is compiled into executable binaries in a corresponding `../bin` subdirectory.

`src/Lib`

A `src/Lib` subdirectory contains code that is compiled into libraries in a corresponding `../../lib` directory. It also contains header files that are copied into `../../include` as part of the build process.

`src/Firmware`

A `src/Firmware` subdirectory contains code that is compiled into a standalone image that is typically stored in the non-volatile memory of a line card (e.g., the flash EEPROM of the IXP1200 EEB.)

`src/Boot`

A `src/Boot` subdirectory contains code that is compiled into a standalone image that is typically loaded into a line card as part of the initialization process performed by the device driver during module installation.

`lib`

A `lib` subdirectory contains subroutine libraries. The corresponding header files are in `../include`.

`bin`

A `bin` subdirectory contains compiled, executable programs.

Pentium Implementation

We implemented a unified device driver for the PMC694 and the IXP1200 EEB in the form of a Linux kernel module. By choosing to develop a Linux kernel module, our device driver will also work with the communication-oriented operating system, Scout [41], as well as the Linux kernel module version of Scout called SILK [3]. Figure 3.11 shows the driver instantiated with one IXP1200 EEB and two PMC694 boards. In this router configuration, there are a total of twelve 100 Mbps Ethernet ports. There are four types of software interfaces to the driver:

Control Plane Interface: The driver exports a `/dev/vera` character device. Through its **ioctl** interface, this is used to perform updates from user space that do not apply to a specific hardware device or network interface (e.g., routing table updates made via the classification hierarchy).

Kernel Interface: The module exports symbols that support dynamically loaded kernel module extensions that extend the **ioctl** interfaces of the `/dev/vera` and `/dev/vera`*N* devices.

Virtualized Network Interfaces: A key element of the module is that the physical network interfaces are *virtualized* as `vth`*N*. Because we want the intelligent line cards to route the packets directly between hardware devices (either within a board or from board to board), many packets will never arrive at the network driver interface to the Linux kernel. However, packets which do arrive on a particular port and which are to be handled by the kernel are sent to their corresponding virtual interface. In this way, packets that are not handled by VERA can be processed normally by the Linux kernel.

Device Interfaces: When the module is instantiated, a character device of the form `/dev/vera`*N* is assigned to each PMC694 or IXP1200 EEB device. This interface allows the boards to be initialized, code to be downloaded, and gives access to the memory and registers of each board.

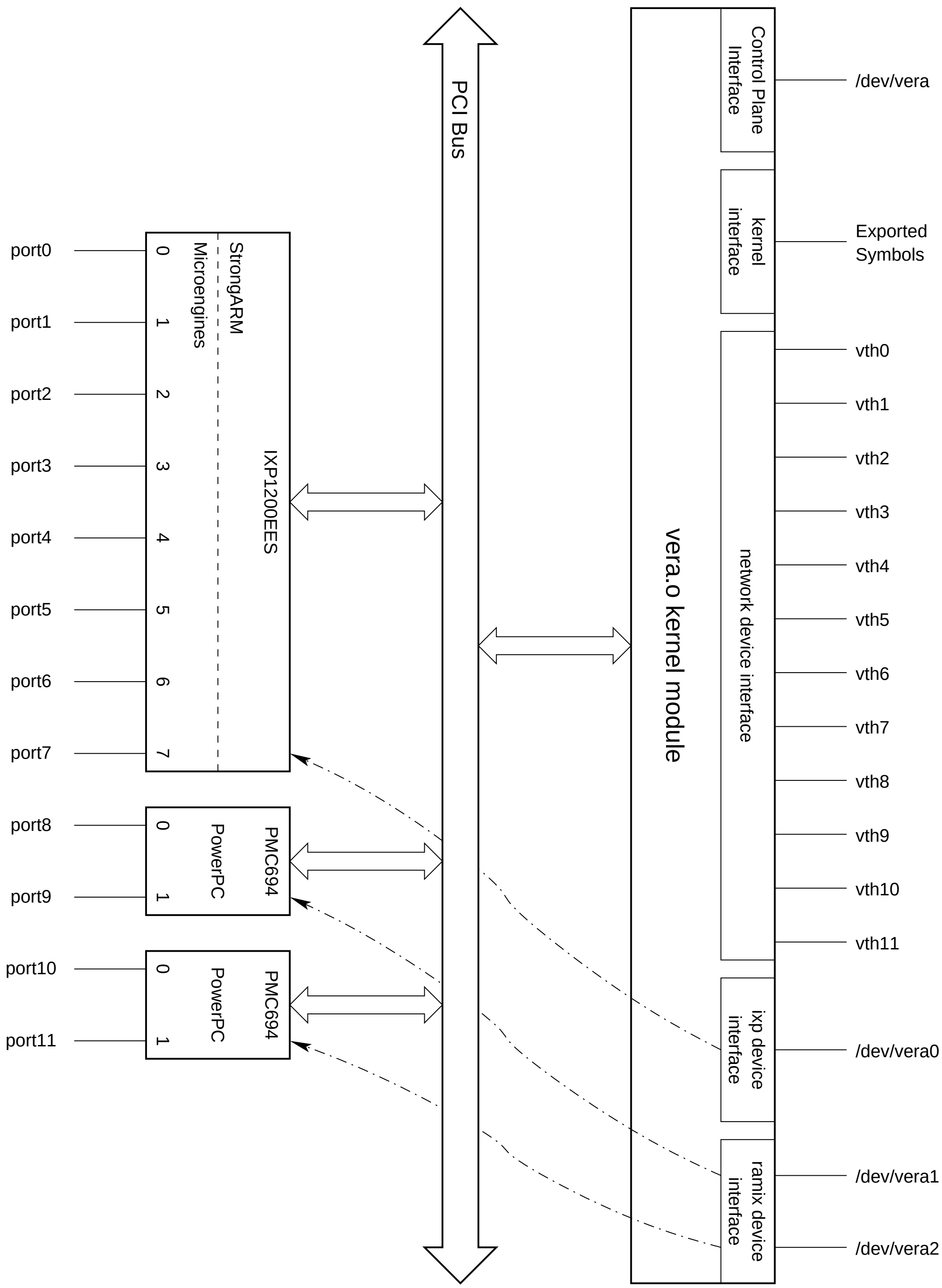

Figure 3.11: The `vera.o` kernel module. In this figure, the module has been instantiated in a system containing one IXP1200 EEB and two PMC694 boards.

3.2.8 Line card Runtime Environment

Firmware / Boot code

Both the IXP1200 EEB and PMC694 line cards needed some customization to work with the Linux device driver in a commodity PC system. As shipped from the factory, the IXP1200 EEB is configured to operate as a stand-alone system in a supplied passive (i.e., no Pentium) PCI backplane connected to a simple 100 Mbps Ethernet line card. The supplied firmware uses this attached line card to download the application from a remote source. Because we want to use the IXP1200 EEB in an active (i.e., with a Pentium) PCI motherboard, we had to reconfigure the board. Specifically, we changed the board jumpers so that it neither generated the global PCI reset signal at power-up nor acted as the PCI arbiter—both of these functions are performed by motherboards. We also wrote our own firmware (and programmed it into the on-board Flash EEPROM) which assumes that the board is installed in a PC—on power-up it requests a region of the PCI bus address space from the BIOS, opens a window from the PCI bus to the SDRAM so that the Pentium can download code directly onto the board. The firmware jumps to an entry point in this downloaded code when it receives a signal from the Pentium.

Because the PMC694 is designed to work with an active PCI motherboard, we did not need to modify the firmware on the board. However, the supplied protocol for downloading application code to the PowerPC memory space involves a handshake for every 32-bit word transferred (rather inefficient) and requires that the application start execution at a particularly inconvenient memory address (in the middle of the address space, splitting the space in two). To alleviate these issues, we created a small boot loader that the driver installs using the supplied protocol. The boot code then copies itself to the high end of the address space and operates similarly to the IXP1200 EEB firmware. That is, it allows the Pentium to write application code anywhere in the address space and then waits for a signal from the Pentium to jump to a specified entry point. Figure 3.12 illustrates how the `sgo` tool loads and runs an executable image on the IXP1200.

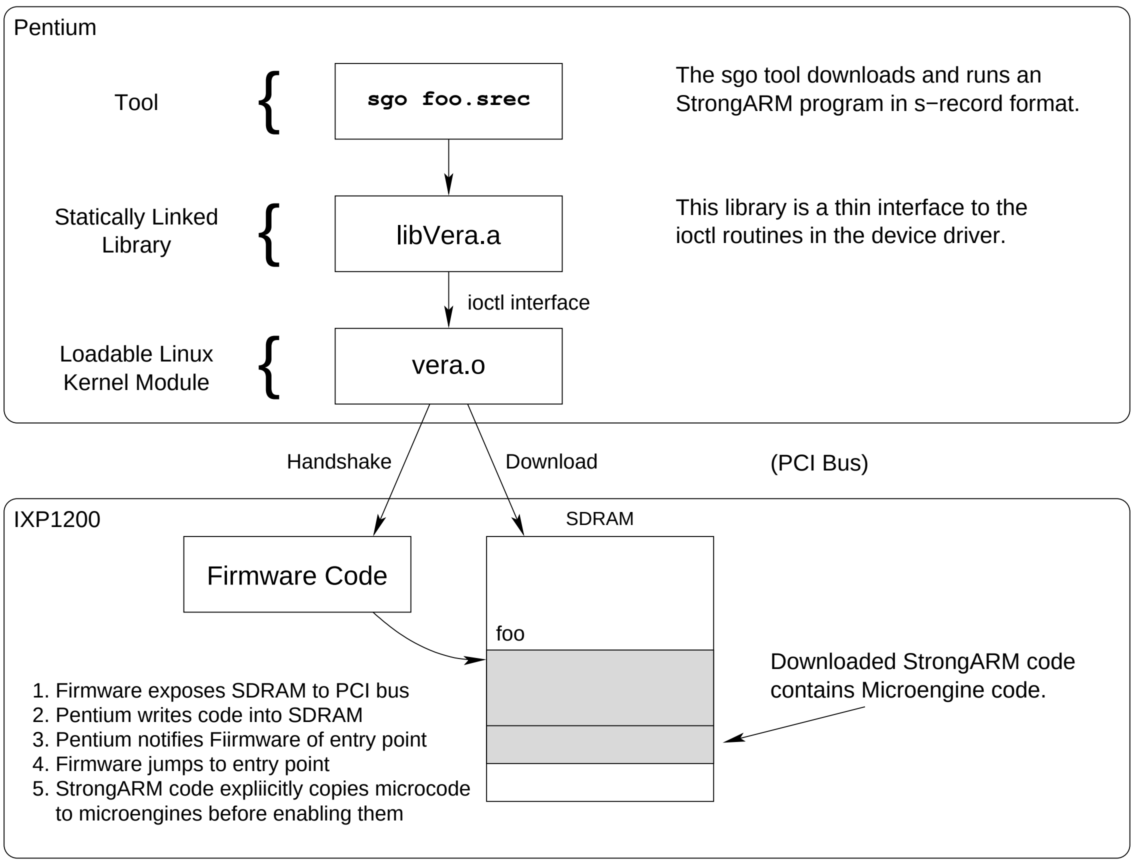

Figure 3.12: Running an executable image on the IXP1200 EEB.

Thread Scheduling

The scheduler running in each scheduling domain of the general purpose processors of the line cards (e.g., StrongARM and PowerPC) builds upon existing scheduling work [51], which in turn, is derived from the WF^{2}Q+ scheduler [4]. We have added hooks to support coarse grain influence from the master processor. In brief, the scheduler assigns a processor share to each forwarder according the the *F_parms* passed to the **createPath** operation. Processor shares are also assigned to the scheduler and classifier threads. The share-based scheduler is augmented to use information about the states of the queues to determine when a forwarder or output scheduler thread should run. In the case of an output scheduler, packets in the queue are assumed to be in local memory so that when an output scheduler thread is activated it will not immediately block as it attempts to fetch the packet data from a remote processor. This helps keep the output link full.

3.2.9 Microengine Environment

When the IXP1200 comes out of device reset, the microengines remain in a reset state until the StrongARM releases them, via a register write, where they then start to execute code in their instruction store. The StrongARM has write access to the microengine instruction store. As part of the initialization of an application, the StrongARM must initialize the microengine instruction store and release the microengines from their reset state.

Virtual Router Processor

Our approach to installing useful packet processing at the microengine level of the processor hierarchy is to statically allocate the microengines to two tasks: (1) a *router infrastructure* (RI) that is able to forward minimum-sized packets at line speed, and (2) a *virtual router processor* (VRP) that runs additional code on behalf of each packet [55]. Since it is impossible to fully predict packet traffic or arrival times, for the sake of robustness we must assume that packets arrive at line speed, and so we statically allocate enough microengine contexts to run the input loop.

The alternative to this static thread scheduling is to dynamically allocate threads to packets. We reject this approach for two reasons: (1) the design of the microengines is not conducive to such an approach—significant processing cycles would be wasted in signaling and coordinating the microengine resources to the tasks, and (2) to ensure we can accept and classify all packets under worst-case conditions.

3.3 Evaluation

In Chapter 2, we concluded that our architecture is extensible and it is robust under the assumption that there is a robust implementation. To show that our implementation is extensible, we need to show that there are CPU cycles available to run new services. Recall from Chapter 1 that our definition of robustness consists of two parts: (1) routers must be able to read and classify packets at line speed and (2) routers must honor the processing guarantees they make. We show this with a VRP that can read and classify packets at line speed and still have resources remaining to run services.

The remainder of this section is organized as follows. In the next three sections we characterize the PCI performance (Section 3.3.1), packet transfer performance (Section 3.3.2, and VRP performance (Section 3.3.3). The performance characterization is used to establish extensibility (Section 3.3.4) and robustness (Section 3.3.5). In addition, the performance characterization is used to set resource limits for the resource allocation methods we introduce in Chapter 4.

3.3.1 Performance Characterization — PCI

Our PCI micro-benchmarks measure PCI bandwidth between the line cards and the Pentium for the testbed shown in Figure 3.1. In these experiments, the Pentium is running a Linux 2.2 kernel, the `vera.o` kernel module, and an extension module running the benchmark application code. The StrongARM and PowerPC processors on the line cards run a minimal VERA environment linked to the benchmark application code.

Tables 3.1 and 3.2 present the raw data from our experiments in copying 64-byte and 1500-byte packets across the PCI bus. In these tables, the *Bus Master* and *Direction* columns describe which device is performing which type of bus cycle. For example, a *host/push* indicates that the Pentium is acting as the bus master and is performing PCI write cycles to "push" the data from the Pentium to the line card. The *Mode* column indicates whether the bus master is using its DMA engine or programmed I/O (PIO). In the case of programmed I/O, we measured the PCI bandwidth when the inner loop copied bytes and 32-bit words (matching the width of the PCI bus). We also unrolled the 32-bit word inner loop so that we copied 64-bits and 128-bits per iteration. Finally, we used the finely tuned **memcpy** routine from the GNU compiler in place of PIO inner loop.

Observations

As expected, byte copies perform the worst. This is because we are only using 8 out of 32 of the data lines for each PCI bus transaction. Increasing the transfer size to 32-bits (matching the PCI bus width) made a significant improvement in all cases. In the case where the Pentium was the bus master (pull only) and the case where the PMC694 was the bus master (push or pull) the improvement was approximately a factor of four. The other cases showed and even more significant improvement. As a bus master, the Pentium was 5.7 and 6.6 times faster when writing to the PMC694 and IXP1200 EEB, respectively. As a bus master, the IXP1200 was 5.4 times faster writing to and 6.7 times faster reading from the motherboard memory. In the cases where the improvement was more than the

Bus Master	Transfer Size	Mode	Direction	64-Byte Packets		1500-Byte Packets	
				Kpps	Mbyte/sec	Kpps	Mbyte/sec
host	8 bits	PIO	push	161.5	10.33	6.82	10.23
host	32 bits	PIO	push	919.9	58.88	39.37	59.06
host	64 bits	PIO	push	975.0	62.40	41.58	62.37
host	128 bits	PIO	push	962.4	61.60	41.53	62.30
host	(memcpy)	PIO	push	1073.4	68.70	44.29	66.44
host	8 bits	PIO	pull	15.5	0.99	0.66	0.99
host	32 bits	PIO	pull	61.1	3.91	2.61	3.91
host	64 bits	PIO	pull	61.8	3.95	2.63	3.95
host	128 bits	PIO	pull	61.9	3.96	2.64	3.95
host	(memcpy)	PIO	pull	62.2	3.98	2.66	3.99
card	8 bits	PIO	push	87.5	5.60	3.69	5.53
card	32 bits	PIO	push	365.5	23.39	14.84	22.26
card	64 bits	PIO	push	363.4	23.26	14.81	22.21
card	128 bits	PIO	push	363.9	23.29	14.81	22.21
card	—	DMA	push	534.2	34.19	17.57	26.35
card	8 bits	PIO	pull	13.4	0.86	0.57	0.86
card	32 bits	PIO	pull	53.7	3.44	2.29	3.43
card	64 bits	PIO	pull	53.8	3.44	2.28	3.41
card	128 bits	PIO	pull	53.8	3.44	2.28	3.41
card	—	DMA	pull	354.1	22.66	15.74	23.61

Table 3.1: Raw PCI Transfer Rates Between the PMC694 (card) and the Pentium III Motherboard (host).

factor of four suggested by the ratio of data transfer size, the PCI hardware is combining successive accesses into more efficient burst cycles.

When unrolling the PIO loop by a factor of two (64 bit transfer size), we measured a significant (greater than 10%) improvement over 32-bit transfers in the case where the IXP1200 was writing data to the motherboard memory. This is most likely due to the fact that by reducing the software overhead, the code is able to feed the PCI interface fast enough to give it more opportunities to combine adjacent accesses into bursts. We measured slight, but not significant, improvements when unrolling the loop to a 128-

Bus Master	Transfer Size	Mode	Direction	64-Byte Packets		1500-Byte Packets	
				Kpps	Mbyte/sec	Kpps	Mbyte/sec
host	8 bits	PIO	push	157.2	10.06	6.7	10.01
host	32 bits	PIO	push	1030.2	65.93	44.2	66.34
host	64 bits	PIO	push	1028.9	65.85	44.1	66.16
host	128 bits	PIO	push	1030.1	65.93	44.0	66.14
host	(memcpy)	PIO	push	1073.9	68.73	44.2	66.35
host	8 bits	PIO	pull	15.3	0.98	0.7	1.02
host	32 bits	PIO	pull	63.7	4.08	2.7	4.05
host	64 bits	PIO	pull	63.7	4.08	2.7	4.04
host	128 bits	PIO	pull	64.3	4.11	2.7	4.09
host	(memcpy)	PIO	pull	64.4	4.13	2.7	4.12
card	8 bits	PIO	push	65.2	4.17	2.8	4.15
card	32 bits	PIO	push	349.2	22.35	14.7	22.12
card	64 bits	PIO	push	387.1	24.78	16.5	24.81
card	128 bits	PIO	push	387.1	24.78	16.5	24.81
card	—	DMA	push	204.7	13.10	32.7	49.01
card	8 bits	PIO	pull	10.3	0.66	0.4	0.66
card	32 bits	PIO	pull	69.1	4.43	2.9	4.42
card	64 bits	PIO	pull	69.8	4.47	3.0	4.46
card	128 bits	PIO	pull	71.1	4.55	3.0	4.53
card	—	DMA	pull	179.2	11.47	16.4	24.66

Table 3.2: Raw PCI Transfer Rates Between the IXP1200 EEB (card) and the Pentium III Motherboard (host).

bit transfer size. With the Pentium as the bus master, we observed that using the tuned **memcpy** library function to write to the PMC694 performed more than 10% better than our simple 32-bit loop.

In most cases, using the DMA engine on the line cards showed significant improvement over PIO. The notable exception is that the DMA engine was slower than PIO for writes performed by the IXP1200 EEB. This is due to the fact that there is overhead in setting up the DMA engine. Linearly interpolating the DMA and 64-bit PIO curves and finding their intersection shows that the cross-over point occurs when the packet size is 166 bytes. (See Appendix A for the calculation of `sa2pt_pio_max`.)

Summary

From our experiments, we see that the highest PCI bandwidth occurs when we use write cycles (i.e., "push"). The best performance for the Pentium occurs when we use the **memcpy** routine. For the PMC694, we should always use the DMA engine to move packets. For the IXP1200 EEB, we should use the DMA engine to read (i.e., "pull") packets of any size and to write packets greater then 166 bytes; for smaller packets, the IXP1200 EEB should use a programmed I/O software loop unrolled to move 64 bits per iteration.

3.3.2 Performance Characterization — Packet Transfer

This section summarizes the results of our experiments with the IXP1200 EEB as part of our study on network processors [55]. These micro-benchmarks measure how fast packets can be forwarded *between* levels of the processor hierarchy. We conclude this section with comments about the packet transfer performance of the PMC694.

Ports to Microengines

We initially measured the system using the eight 100 Mbps Ethernet ports on the IXP1200 EEB fed by eight Kingston KNE100TX PCI Ethernet cards (based on the 21143 "Tulip" chip) as traffic sources. (A pair of these cards are installed in each of four 450 MHz Pentium IIs running packet generator software.) When configured to generate minimum-sized (64-byte) packets, each card transmits at 141 Kpps, which is 95% of the theoretical maximum of 148.8 Kpps [31]. With these traffic sources, the microengines are able to sustain line speed across all eight ports resulting in a measured aggregate forwarding rate of 1.128 Mpps. This is an expected result as the theoretical forwarding capacity of the processing and memory resources in the IXP1200 are much greater than the 800 Mbps of testbed traffic.

In an additional experiment, we measured the maximum throughput of the microengines running a null VRP (i.e., no loaded services) at a rate of 3.47 Mpps [55]. This measurement is independent of the number of ports and reinforces the fact that the microengines have sufficient processing bandwidth to support eight 100 Mbps ports at line speed.

Microengine to/from StrongARM

Since the StrongARM and the microengines share the SDRAM and SRAM, packets need not be copied to "move" them between the StrongARM and the microengines. The only latency is the cost of a microengine signaling the StrongARM to inform it that a packet is available.

Upon detecting that a packet requires service by the StrongARM (e.g., there is a miss in the route cache or the packet contains IP options), a microengine input context enqueues the packet in a StrongARM-specific queue instead of a queue assigned to an output port. At this point, we have two options for signaling the StrongARM: interrupt the StrongARM or let the StrongARM poll to see if any packets have arrived. In both cases, the StrongARM dequeues the next packet from this queue, performs whatever processing is required, and places the packet on the appropriate output queue.

We measured the maximum rate that the StrongARM can process packets by having it run a null forwarder, with the microengines programmed to pass all their packets to the StrongARM. With this configuration, we achieve a maximum forwarding rate of 526 Kpps using a polling loop on the StrongARM; interrupting the StrongARM was significantly slower. By adding a delay loop (in the StrongARM polling loop) that counts to a pre-determined value, we can determine how many extra cycles we can insert before impacting the forwarding rate. In our system we found that any delay reduced the forwarding rate and conclude that the StrongARM has no additional cycles available to compute on packets when receiving them at this rate.

IXP1200 to/from the Pentium

We move packets between the IXP1200 and the Pentium over the PCI bus. Our implementation uses the IXP1200's DMA engine plus I_2O queue management hardware registers. We measured the maximum rate that the Pentium can process packets by having it run a loop that reads packets from a queue on the IXP1200, and then writes the packet back onto a queue on the the IXP1200. This experiment is an extension of our PCI micro-benchmarks presenting in the previous section. In addition to moving packets, this experiment accesses the I_2O registers to determine the buffer addresses. (Due to a silicon error, the I_2O mechanism does not work on the B version of the IXP1200. In our experiment, we used an *a priori* sequence of buffer addresses and had the Pentium perform additional PCI reads and writes to a memory location on the IXP1200 EEB to introduce the same delay as accessing the I_2O registers on the IXP1200.) The StrongARM is programmed to feed packets to the Pentium as fast as possible. We also inserted a delay loop on both sides to determine the number of spare cycles available, that is, cycles not involved in the data transfer. The results are given in Table 3.3, which shows that the router is able to forward up to 534 Kpps through the Pentium. This rate saturates the StrongARM, but leaves 500 cycles per packet available on the Pentium.

Packet Size (Bytes)	Rate (Kpps)	Pentium (Cycles)	StrongARM (Cycles)
64	534.0	500	0
1500	43.6	800	4200

Table 3.3: Measured Maximum Forwarding Rate and Excess Per-Packet Processor Cycles from the IXP1200 EEB to the Pentium and back.

Note that up to this point we have focused on 64-byte packets. This is because processing minimal-sized packets is the worst-case scenario in the sense that it represents the highest packet rate. It is also the case that forwarding larger packets scales linearly on the microengines: forwarding a 1500-byte packet involves forwarding twenty-four 64-byte MPs. Crossing the PCI bus is different, however, since the DMA engine runs concurrently with the StrongARM. Also note that when a 1500-byte packet does arrive, we will first move the smaller IRH across the PCI bus; the entire packet will be moved only when necessary. Our experiments move the entire packet.

PMC694 Comments

Because we did not enter into a non-disclosure agreement (NDA) with Intel, we did not have access to the datasheets and were not able to develop a polling device driver for the 82559ER MAC chips on the PMC694 line card. However, based on the PCI micro-benchmark results in Table 3.1, we can estimate the capability of the PMC694. For these estimates, we assume that the 82559ER MAC chip can operate continuously at line speed, is able to push data into the PowerPC memory as efficiently as the Pentium, and is able to accept data pushed by the PowerPC at the PowerPC's DMA rate.

Ports to PowerPC

To estimate how many packets can be moved from the ports to the PowerPC and back to the ports, we first calculate that maximum packet rate and then identify any bottlenecks. For the PMC694, the maximum packet (receive) rate is 2×148.8 Kpps, or 297.6 Kpps. This means that a new packet may arrive every 3.360μs. A 64-byte packet is encapsulated in a 72-byte Ethernet frame. If the MAC pushes the packet to the PowerPC at 68.7 Mbyte/sec, this will take 1.048μs. If the PowerPC pushes the packet back to the MAC at 34.19 Mbyte/sec, this will take 2.106μs. This leaves an idle time of 0.206μs before the next packet arrives on one of the MACs. Because there is idle time on the bus, we conclude that there is sufficient switching bandwidth to move packets from the ports through the PowerPC and back at the aggregate line rate of 297.6 Kpps. In addition, because the PowerPC is not using PIO to move the data, we can pipeline the system and have $3.360\mu s \times 266$ MHz, or 893 cycles of processing available per packet (less DMA setup and other overhead). Table 3.4 summarizes these calculations and gives packet rate and available CPU cycles for the 1500-byte packet case.

Packet Size (Bytes)	Ethernet Frame Size (Bytes)	Packet Rate (Kpps)	PowerPC (Cycles)
64	72	297.6	893
1500	1508	8.22	32300

Table 3.4: Estimated Maximum Forwarding Rate and Excess Per-Packet Processor Cycles from the ports to the PowerPC and back.

PowerPC to/from Pentium

As we have just shown, the PCI bus has sufficient bandwidth to support two 100 Mbps Ethernet ports. Table 3.5 gives estimates of the number of available PowerPC and Pentium cycles when sending packets from the PowerPC to the Pentium and back based on the PCI micro-benchmarks of Table 3.1. Because the packet rates are limited by the speed of the ports, the PowerPC cycles are the same as Table 3.4. The Pentium cycles are estimated by subtracting the Pentium's PIO transfer time from the packet arrival period and converting to cycles for a 733 MHz processor.

Packet Size (Bytes)	Ethernet Frame Size (Bytes)	Packet Rate (Kpps)	Pentium (Cycles)	PowerPC (Cycles)
64	72	297.6	1460	893
1500	1508	8.22	72560	32300

Table 3.5: Estimated Maximum Forwarding Rate and Excess Per-Packet Processor Cycles from the PowerPC to the Pentium and back.

3.3.3 Performance Characterization — VRP

To be extensible, a router must be able to operate at line speed and still have processing resources available to run new services. From the experiments in the previous section, we know that the microengines can forward 3.47 Mpps using a null VRP. Note that this is greater than the worst-case aggregate line speed of 1.128 Mpps for the IXP1200 EEB system. Based on experimentation [55], we measured that a VRP with the following characteristics can be applied to every packet and maintain the worst-case aggregate line speed of 1.128 Mpps.

- Each packet is fragmented into 64-byte pieces and becomes accessible to the VRP in registers one fragment at a time. The first fragment holds both the TCP and IP headers.
- In addition to the registers that hold each fragment, the code running in the VRP has access to 8 general purpose 32-bit registers. Values stored in these registers

are not maintained across invocations of the VRP and, therefore, can only be used for temporary state (e.g. intermediate computational results or state loaded from SRAM). An additional register contains the SRAM address of the flow-specific state.

- The VRP code can execute up to 240 cycles worth of instructions.
- The VRP code can perform up to 24 SRAM transfers (reads or writes) of 4 bytes each.
- The VRP code can perform 3 hashes with support of the hardware hashing unit of the IXP1200.

Note that the VRP budget depends on the hardware configuration. If the IXP1200 must support more ports, then there will be fewer resources available to the VRP. Similarly, if there are fewer ports, then there will be more resources available to the VRP.

3.3.4 Extensibility

To show that our implementations are extensible, we must show that there are cycles available to apply to the extensions under worst-case conditions. For the IXP1200 EEB implementation, the results of our experiments in the previous section show that the microengines can apply 240 cycles of processing to every packet at the aggregate line speed of the eight ports on the line card. For the PMC694 implementation, we argue that based on the PCI micro-benchmarks of Section 3.3.1 the PowerPC can apply approximately 800 cycles of processing to every packet at the aggregate line speed of the two ports on the line card.

Because we have cycles to apply to extensions under worst-case conditions, we conclude that our implementations are extensible. Note that we have not explicitly mapped the services to the processors; we have only shown that there are cycles available. In the next chapter, we address the problem of mapping the services onto the various processing elements of the router.

3.3.5 Robustness

Recall from Chapter 1 that our definition of robustness consists of two parts: routers (1) must be able to read and classify packets at line speed and (2) must honor the processing guarantees they make. We will address these points here. Note that an important implication of this definition is that the router does not take on more work than it can handle. This separable problem is addressed in Chapter 4.

We start by considering the router of Figure 1.4 (page 14). Figure 3.13 depicts three possible switching paths for this router. Path A includes the microengines and the IXP1200 memory. Path B includes the microengines, the IXP1200 memory, and the StrongARM. Path C includes the microengines, the IXP1200 memory, the StrongARM, the Pentium memory, and the Pentium. The shaded boxes represent processors, while the non-shaded boxes represent memory that implements packet queues and buffers.

To validate the ability of the system to read and classify packets at line speed, we configured the microengines to run a synthetic suite of forwarders[4] that utilize the full VRP budget. Under an offered load of 1.128 Mpps (the aggregate line speed of the ports), the microengines were able to forward all packets along path A. In a second experiment, we directed the microengines to classify an increasing percentage of the packets as exceptional, thereby simulating a flood of control packets. These exceptional

[4]based on those in Table 4.1 (page 99)

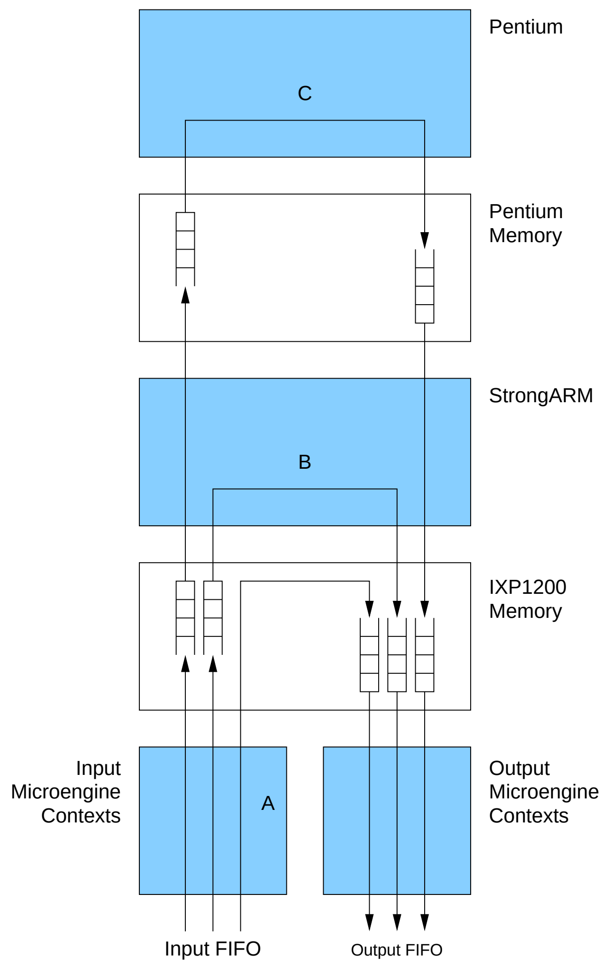

Figure 3.13: Three switching paths through the Pentium/IXP1200 processor hierarchy.

packets were directed to the StrongARM and had no effect on the router's ability to forward regular packets (those whose forwarder was installed on the microengines)—the microengines were able to sustain the aggregate line rate of the ports.

Because our system can handle packets at line-speed under worst-case conditions and provide isolation between services through static scheduling, we conclude that our system meets the first criteria for robustness. Note that this does not mean that our system is so over-provisioned that we can apply millions of cycles to every packet. It means that we can make an initial classification decision for every packet (possibly deciding to drop the packet) and still have a cycle budget to apply to each packet.

To be robust, the router must also ensure that it does not take on more work than it can guarantee to finish. That is, there must be a form of service admission control to ensure that the router is not over-burdened. In the next chapter, we address the admission control problem as a part of the resource allocation problem.

Chapter 4

Resource Allocation

In the previous chapters we have introduced and discussed an extensible router architecture that uses multiple, heterogeneous processors to classify, forward, and schedule packets. Up to now we have not discussed how to allocate resources (processing, switching) within the router. Properly allocating resources is key to maintaining robustness. Recall that in addition to operating at line speed, a robust router must honor the processing guarantees it makes. By admitting a new service, the router is making a guarantee. Admission control addresses the problem of determining which services the router can allow to run. In addition, mapping services onto the hardware is key to providing extensibility. If were are not able to admit *any* services, clearly our router is not extensible.

This chapter first motivates the resource allocation problem and the related admission control problem and then introduces some of the issues specific to our architecture. Next, we describe several algorithms for placing services on the processors of the router. Finally, we evaluate these algorithms against anticipated workloads.

4.1 Problem Space

This chapter investigates the following mapping problem: given a hardware configuration containing multiple processing elements, how do we best place the services (that is, the functions implementing the services) on these underlying processing elements. The problem is similar to job placement on multicomputers [18] except there are the additional complexities that (1) packets must traverse one processor to reach another, and (2) the processors are heterogeneous and may have widely varying characteristics.

Note that the services are variable—a given router can support a different number of services, each service can have different computational requirements, and some services apply a variable number of cycles to each packet (e.g., they may be data-dependent). In contrast, the classification and scheduling stages, whether implemented in hardware or software, can be viewed as part of the router's fixed infrastructure—each is associated with a particular port, takes a bounded amount of time for each packet, and must run fast enough to input/output packets at line speed.

4.1.1 Motivating Example

Based on our experiments of our implementation depicted in Figure 3.13, the router can forward packets at a maximum rate of 3.47 Mpps along path A, 526 Kpps along path B, and 310 Kpps along path C [55], but there are three caveats that affect our decision as to where to place a function implementing a particular service.

First, we cannot simultaneously support paths B and C at their maximum rates since the StrongARM is involved in both. If we give priority to packets destined for the Pentium and let the StrongARM primarily serve as a bridge between the microengines and the Pentium, we limit the services that run on the StrongARM (corresponding to path B) to those that fit within its remaining capacity. Similar interference between path A and paths B and/or C is possible, except that in our design the work for a microengine context to pass a packet up to the StrongARM is the same as the work to implement path A; no additional cycles are required.

Second, more complicated services require more cycles-per-packet (cpp), possibly reducing the maximum forwarding rate. In the case of the Pentium, we have 1510 cpp available at the maximum 310 Kpps rate; more expensive services will obviously lessen the sustainable forwarding rate. In the case of the microengines, all of the available capacity is needed to achieve the 3.47 Mpps forwarding rate. If we have an aggregate line speed below this rate, then there will be excess capacity that can be used to run additional services. Note that the aggregate rate of eight 100 Mbps ports is 1.128 Mpps—well within the 3.47 Mpps capacity of the microengines of an IXP1200. Recall from the last chapter that we sized the VRP to a static budget of 240 cycles and 24 SRAM accesses per packet when the microengines are handling eight ports.

Deciding which services to run on the StrongARM is complicated by the fact that the StrongARM must support the Pentium (as a bridge) and because it shares SRAM and DRAM bandwidth with the microengines. This means an arbitrary service running on the StrongARM has the potential to interfere with the microengine's ability to forward packets at line speed. As a consequence, the StrongARM must be included in the same memory resource budget as the microengines.

Third, even though we know the maximum rates that can be supported by the Pentium and StrongARM, in the worst case all arriving packets require more processing than the microengines can provide, and so have to be passed up the processor hierarchy. This means the higher levels of the processor hierarchy must differentiate among packets based on classification done at the microengine level and then schedule their available capacity in some meaningful way.

4.1.2 Admission Control and Placement

Service placement is a generalization of admission control. In fact, admission control is a consequence of service placement. That is, the system first creates a list of processors and switching paths that can support the service. If the list is empty, the service is not admitted—the router does not admit services that will jeopardize robustness. Note that the system can include additional policy restrictions (e.g., user A may not install services on processor B) in its admission control and placement logic.

The hardware abstraction layer maintains a database of the component capabilities. The capabilities include the bandwidth of the ports and switches as well as the processing cycles available to the upper software layers. Specifically, this information is used by the OS to determine appropriate placement of services. It is important to note that the HAL takes overhead processing cycles and switching capacity into account.

The admission control mechanism must also decide how many services to allow on the Pentium. For each such service, the requester specifies the expected packet rate and the expected number of cycles expended on each packet. From these two values, the mechanism determines the service's total cycle rate. The service can be admitted only if the processor has sufficient cycles-per-second available and the total packet rate remains below the maximum that the Pentium can sustain. Admission control to the Pentium, as well as the strategy for scheduling the Pentium's cycles, are discussed in Qie, Bavier, Peterson, and Karlin [51].

4.2 Hardware Performance Model

As shown in Chapter 2, our hardware model consists of three major components: *processors*, *switches*, and *ports*. A router topology is the connected graph of the *direct* connections among these components. Services can be placed on any processor in the topology, but packets being handled by a particular service must traverse a switching path through the topology that includes the processor on which that service is placed.

4.2.1 Processors

Each processor in the system is parameterized by the following performance metrics:

scheduling type: Either *static* or *dynamic*. Statically scheduled processors require that all services fit assuming worst case processing rates and worst case bandwidth rates. Dynamically scheduled processors use average service rates and average packet rates to determine if a service will fit on a given processor. Whether statically or dynamically scheduled, we assume that all the services to the system have an independent impact (see Chapter 2)—they do not interfere with each other and the cost each service imposes on the system is additive. As demonstrated elsewhere, this property can be enforced on both statically [55] and dynamically [51] scheduled processors. The scheduling type determines how to interpret the next parameter, cycle rate.

cycle rate: For statically scheduled processors, this is the number of available processing cycles per packet. For dynamically scheduled processors, this is the total number of available processing cycles per second.

transmit cost: This is the cost in processor cycles to transmit a packet to a given switch. It is expressed as a combination of cycles/packet and cycles/byte. The cycles/packet overhead is independent of the switch speed and independent of the packet destination. For processors with DMA engines, the cycles/packet will be approximately zero. For processors that use programmed I/O (PIO), we assume the existence of a deep enough write buffer to make the transmit cost independent of the switch speed.

receive cost: This is the cost in processor cycles to receive a packet from a given switch. It is expressed as a combination of cycles/packet overhead and cycles/byte.

4.2.2 Switches

Functionally, the switch abstraction provides an interface for interprocessor data movement and *distributed queues* whose head and tail are on different processors. Part of the fixed infrastructure running on each processor implements each queue's head and tail. This corresponds to the classifier that receives packets and decides how to process them (queue head), and the scheduler that selects a packet and then transmits it (queue tail). The performance of these two activities is captured, respectively, by the **transmit cost** and **receive cost** parameters defined above.

The performance of the switch itself is characterized by by both its bandwidth and overhead (e.g., DMA setup, the address phase on the PCI bus, the interframe gaps on an Ethernet). Thus, each switch in the system is parameterized by the following performance metric:

transfer time: The time it takes to transfer a packet from point A to point B through the switch. It is specified with the time to transfer a minimum-sized packet and the time to transfer a maximum-sized packet. Intermediate sizes are computed using linear interpolation. Switches are not assumed to have symmetric bandwidth between the same two processors. This is why we characterize the directed transfer time. Also, when computing the capacity of a switching path one must take into account that when a packet is transmitted slowly, it is the same as transferring a larger packet more quickly—it takes the same "wire time." Finally, a given switch will have different performance characteristics depending on the particular processors or ports to which it is connected.

4.2.3 Ports

Ports essentially serve as packet producers/consumers in our model. They are trivially parameterized by the following performance metrics:

duplex: Either half-duplex (the sum of the input and output bandwidth is the line-speed of the port) or full-duplex (independently, the input bandwidth and the output bandwidth is the line-speed of the port). Full-duplex gives twice the bandwidth of half-duplex.

packet time: The overhead time on the wire for a "zero-length" packet. In addition to the overhead fields in the Ethernet frame, this parameter captures the required interframe gap between packets.

byte time: The time on the wire for each byte in the packet.

4.3 Workload

Determining a representative set of services is problematic since there is no well-established extensible router workload. Thus, our approach is to define a plausible workload, based on a characterization of the limited set of services we have implemented, and then vary this workload in an effort to understand the robustness of the algorithms.

We have implemented six example services that fit on the microengines. Table 4.1 gives the memory and cycle requirements for each. The first (TCP splicer) connects two TCP connections in a cut-through path. The second (Wavelet Dropper) thins a wavelet-encoded video data stream. The third (ACK Monitor) watches a TCP connection for repeat ACKs in an effort to determine the connection's behavior [45]. The fourth (SYN Monitor) counts the rate of SYN packets in an effort to detect a SYN attack. Port Filter is a simple filter that drops packets addressed to a set of up to five port ranges. The last is minimal IP processing (IP--), which consists of decrementing the TTL, recomputing the checksum and replacing the Ethernet header. (Note that the IP header also needs to be validated—the checksum verified and the version and length fields checked—but this is done as part of the classifier rather than the forwarder.)

Forwarder	SRAM Read/Write (bytes)	Register Operations (instructions)	Registers Needed
TCP Splicer	24	45	7
Wavelet Dropper	8	28	4
ACK Monitor	12	15	4
SYN Monitor	4	5	0
Port Filter	20	26	2
Minimal IP (IP--)	24	32	2

Table 4.1: Cycle, Memory and Register Requirements of Example Services. From [55].

In addition, we have measured more complicated services such as TCP proxies and full IP to require at least 800 and 660 cycles per packet, respectively. Also, the prefix matching algorithm we use as part of classification [56] requires on average 236 cycles per packet.

A workload is an ordered list of services that are to be installed on our extensible router. The next section describes the parameters of the services. For our evaluations, we randomly generate the parameters for each service based on a distribution of service classes that characterizes the workload. Section 4.3.2 describes the service classes.

4.3.1 Service Characterization

We characterize the processing requirements of services as follows. Note that we decouple the input and output profiles to model both filtering services that discard packets (and hence have more input than output), and multicast services that replicate packets (and hence have more output than input).

processing rate: The average and maximum processing rate for each packet, specified as cycles/packet.

input rate: The average packet rate consumed by the service, specified as packets/sec and bytes/packet.

output rate: The average packet rate produced by the service, specified as packets/sec and bytes/packet.

type: Services are characterized as either *per-flow* or *global*. Each packet is processed by all of the global services on the current processor plus the one per-flow service identified during classification. The input and output rates are left unspecified for global services since they are applied to every packet. The introduction of these two types of services is due to both the nature of the services we expect to support as well as the design of the IXP1200 microengines. We expect to perform some operations on every packet (e.g., decrement TTL, update checksum, strip link-layer header, or count packets). Because branches are expensive on the microengines (the consume both additional cycles and space in the instruction store), we want to optimize the application of these global services that the program counter simply falls through to each global service.

In general, the input rate and output rate are distinct. For our evaluations, we simplify the evaluation by assuming that the packet rates into and out of a given service are equal.

4.3.2 Service Classes

We characterize the distribution of processing rates of the services in two stages. First, based on our experience implementing actual services, we define four classes of services. Each of these classes is characterized by an average processing rate, the standard deviation of a normal distribution, and a minimum and maximum cap. The four classes are summarized in Table 4.2. These parameters were chosen based on measurements of actual services on a prototype router.

Second, we vary the workload according to a distribution of the services across these four classes. Our starting point is to assume 9% of the services are **tiny**, 90% are evenly split between **small** and **medium** (i.e., 45% each), and 1% are **large**. We selected this distribution (abbreviated 9/45/45/1) because it roughly matches the capability of the hardware—the architectures we are considering are ill-suited for workloads that need to support more than a handful of very large applications (such as a traditional scalable

Class	Processing Rate Distribution (cpp) Avg	S.D.	Min	Max	Description
Tiny	50	25	10	100	These services are global (i.e., applied to every packet), and include packet counters and filters.
Small	300	75	150	500	These services are per-flow, only operate on the initial 64 bytes of the packet, and include header editors, redirectors, smart droppers, and packet taggers.
Medium	3500	500	1000	5000	These services are per-flow, and include proxies, content caches, application-specific overlay networks, and active protocols. Some only operate on the initial 64 bytes of the packet while others operate on the entire packet. We elected a mix of 50% each.
Large	50K	15K	5000	100K	These services are per-flow, operate on the entire packet, and include server programs and control protocols (e.g., BGP).

Table 4.2: Workload Forwarding Classes.

server might be asked to support) or more than a few services that must be applied to every packet (there simply are not enough cycles per packet available at high bandwidths). In other words, **small** and **medium** services best fit the sweet-spot of the architecture. They also happen to correspond to majority of the services we have encountered in practice. We evaluate the impact of varying this distribution later in this section.

Given these processing rates, we then choose the packet arrival rates such that they are inversely proportional to the processing rate (i.e., expensive services expect a lower packet arrival rate than those with inexpensive processing rates).[1] We also select packet

[1]The actual ratios are normally distributed around an average arrival rate that is inversely proportional to the processing rate.

sizes from a bi-modal distribution. Packet sizes are chosen (with equal probability) from normal distributions with a mean of either 128 bytes or 1000 bytes. The lower value models small packets such as TCP/IP ACKs while the larger value models packets containing content [43, 44]. We also limit the packet size to the legal Ethernet range of 64–1518 bytes.

4.4 Algorithms

Placing a given service on a router is a two step process. First, we must determine the set of processors where the service will *fit*—that is, not cause any particular resource to exceed a pre-determined, resource-specific threshold. Second, from the set of viable alternatives, we must select which processor (if any) to receive the service. For the algorithms presented here, we do not include policy restrictions (e.g., user A may not install services on processor B); however, our method can be extended in a straightforward way to incorporate this kind of logic.

When we can characterize the resource utilization of a service placed on a particular processor of the router in terms of a fixed overhead, a per packet, and a per byte utilization (and can show that this characterization already takes into account any negative resource impact due to packets contending for resources), we have a system that supports the concept of independent impact.

4.4.1 Utilization Vectors

To support the placement process, we define a d-dimensional *utilization vector*, along with associated manipulation primitives. Each of the d non-negative components of the utilization vector correspond to a particular resource (e.g., Pentium utilization, PCI bus utilization, or StrongARM utilization) in the router. Each component is normalized so that 0 indicates no resource utilization and 1 indicates full resource utilization.

When considering whether a service s can be placed on a particular processor p we use the following utilization vectors:

Router Utilization: This vector, denoted $\mathbf{u}$, is the cumulative utilization of all the services already placed on the router. Initially, this vector is $\mathbf{0}$.

Service Impact: This vector, denoted $\mathbf{i}_{s,p}$, is the incremental utilization of placing a particular service on a particular processor.[2] Because each service has an independent impact on the router, this vector is independent of the router utilization vector.

Provisioning Vector: The components of this static vector, denoted $\mathbf{p}$, set the maximum utilization threshold for each resource. With the origin as one corner, this vector determines the opposite corner of a d-dimensional rectangular parallelepiped. As long as the router utilization remains within this region, the router is not over-utilized. Typically, each component of the provisioning vector is 1 making the region a unit hypercube.

Using these definitions, we say that a service s can be placed on processor p as long as the "sum" of the current router utilization $\mathbf{u}$ and the service impact $\mathbf{i}_{s,p}$ lies within the region defined by the provisioning vector $\mathbf{p}$.

[2]The service impact also depends on the particular path the packets will take through the system. In this thesis, we assume that the path traversing the minimum number of components from the input port, to the processor, to the output port is best. Since we assume a hierarchical organization, determining which components lie on the path is trivial.

Calculating Service Impact

The precise details of how to compute a service's impact depend on what aspects of the router (hardware and software) and the service (requirements) are included in the model. This section gives an overview of some of the considerations; Section 4.5 describes the method of calculation for a specific set of experiments.

Each component of the service impact vector is computed independently based on how the service interacts with the corresponding router resource as follows:

Dynamically-scheduled processors: The CPU resource is modeled in terms of maximum cycles per second. Services that are placed on a dynamically-scheduled processor consume cycles based on the service's *average* processing requirements per packet, average packet rate, and average packet size. Services that are placed so that packets must pass through a dynamically-scheduled processor have an impact on the processor based on the average packet rate and the average packet size. Note that we make the simplifying assumption that global services cannot be placed on dynamically-scheduled processors.

Statically-scheduled processors: The CPU resource is modeled in terms of maximum cycles per packet. All packets are guaranteed this many cycles under all conditions (specifically, when packets are arriving at the processor at the maximum rate possible based on the topology of the router). Services are placed on statically-scheduled processors based on their *worst-case* processing requirements independent of packet rate or packet size. Services that are placed so that packets must pass through a statically-scheduled processor have no impact on the processor. The maximum available cycles per packet already takes into account the fact that processor can move packets at line speed.

Ports and Switches: When a per-flow service is placed on the router, it impacts the utilization of the associated input and output port and all the switches along the path based on the average packet rate and average packet size for the service. Global services have no effect on port or switch utilization.

Utilization Vector Addition

The only restrictions on the definition of "sum" is that it be associative, commutative, and closed; that is, the result of adding a set of utilization vectors is a utilization vector and the result is independent of the order in which they are combined.

Ordinarily, the "sum" of two utilization vectors is defined as the usual component-by-component sum. However, if the router contains statically-scheduled processors supporting both per-flow and global services, the utilization vector and the definition of the "sum" become more complex. For packets associated with a per-flow service hosted by a statically-scheduled processor, the worst-case number of cycles required is the sum of the worst-case requirements for all of the global services hosted by the processor plus the worst-case requirement for the per-flow service. To ensure that this stays within the strict per packet budget of the processor under all conditions, we must separately maintain both the global service contribution (the *sum* of the worst-case global service requirements) and the per-flow service contribution (*maximum* of the worst-case per-flow service requirements). By using three components (global service sum, per-flow service maximum, and their sum) in the utilization vector, we capture the dual nature of these processors. With this scheme, the provisioning vector not only limits the overall processor utilization, it can independently limit the utilization of global services and per-flow services.

4.4.2 Online Placement Algorithms

As mentioned in Chapter 3, we model services as having an independent impact on the resources of the router. The goal of the placement algorithm is to place as many services as possible from a set of services—the *workload*. Broadly, our placement algorithms can be divided into two classes: *online* and *offline*. Online algorithms consider each service in the order specified by the workload while offline algorithms (discussed in Section 4.4.3) simultaneously consider all of the services of the workload. Our service placement algorithms currently impose the constraint that services do not move once they have been placed.

Our expectation is that a deployed router would use a combination of online and offline placement algorithms. The router would use an offline algorithm to place a static set of universal services at boot time and then use an online algorithm to dynamically place services on processors as the requests to place services arrive.

In the following descriptions, the running times of the algorithms are expressed in terms of n, the number of services to place; m, the number of processors; and, d, the dimensionality of the utilization vector.

For each service, considered in workload-order, the online algorithm determines the set of feasible processors—those for which the resulting utilization vector is within the bounds set by the provisioning vector. From this set, the algorithm then chooses one of the processors based on one of the following placement strategies. If the service will not fit on the router, it is skipped and the algorithm continues with the next service in the workload. Because only one pass is made through the workload, these online algorithms take $O(nmd)$-time to process the n services.

Ordered: This online placement strategy is parameterized by an ordered list of processors on the router. A service is placed on the first processor (in order) on which it fits. The name of an instance of the algorithm (e.g., `UE-SA-PT`) is the concatenation of the processor abbreviations (`UE`, microengine; `SA`, StrongARM; `PT`, Pentium; `PP`, PowerPC) in the order considered for placement on the router.

L2-Norm: This placement strategy places each service on the processor that minimizes the L^2 norm (Euclidean norm) of the resulting router utilization vector. Geometrically, vectors with the same L^2 norm lie on the same hypersphere. The intuition is that this strategy keeps the router utilization vector as close as possible to the origin.

Balanced: This placement strategy places each service on the processor that minimizes the L^∞ norm of the resulting router utilization vector. (The L^∞ norm of the utilization vector is the value of its maximum component.) Geometrically, vectors with the same L^∞ norm lie on the same hypercube. The intuition is that this algorithm tends to *balance* the utilization of the components, not letting any one get too far ahead of the others. The balanced strategy allows the router utilization to get farther from the origin than a L^2 norm strategy as long as the utilization vector stays away from the boundaries defined by the provisioning vector.

Hybrid: As the name implies, this placement strategy combines the Ordered and Balanced strategy. It first attempts to place a service on a pre-selected processor. If the service does not fit, the service is placed on one of the remaining processors using the Balanced strategy.

4.4.3 Offline Placement Algorithms

An offline algorithm considers all of the services in the workload simultaneously to determine the best configuration. Because each service could be placed on any of m processors (or not placed at all), the number of possible configurations is $(m+1)^n$. Given that we are considering workloads with hundreds of services, an exhaustive search is not feasible. (An exhaustive search, however, may be reasonable for optimally placing a small set of common services during router boot time.)

As a comparison point and because early experimentation with the online algorithms led us to believe that the order in which services were considered would have an impact on the number which could ultimately be placed on the router, we developed the following offline algorithm:

Scan: The offline **scan** algorithm first creates a list with nm items each containing references to one of the services s from the workload and one of the processors p in the router. This list is then repeatedly *scanned* for the (s, p) pair which, if placed on the router, would yield the minimum L^∞ norm of the resulting router utilization vector, $\mathbf{u}$. After updating $\mathbf{u}$, all other entries in the list containing s are removed (so that a given service is not placed more than once), and the process repeats. This algorithm runs in $O(n^2md)$-time.

4.5 Experimental Results on IXP1200 EEB

This section explores the algorithm space by evaluating the placement algorithms for a specific hardware configuration under varying service workload. Our intuition is to (1) place on the microengines those services that require relatively few processing cycles per packet in the worst case but expect a lot of traffic, (2) place on the Pentium those services that require many processing cycles per packet and expect little traffic, and (3) place on the StrongARM those services with "intermediate" processing and bandwidth requirements.

A "closest-to-the-ports" scheme would give the microengines the highest priority, the StrongARM an intermediate priority, and the Pentium the lowest priority. A "closest-to-the-root" scheme would assign priorities in the opposite order.

Our placement results depend on the characteristics of the router and the workload. We discuss these next.

4.5.1 IXP1200 Router Parameters

We evaluated the placement algorithms using the hardware configuration shown in Figure 1.4 (page 14). It includes a 733 MHz Pentium III PC with a 32 bit, 33 MHz PCI bus connected to an off-the-shelf 200 MHz IXP1200 network processor-based intelligent line card with eight 100 Mbps Ethernet ports.

The router is modeled with both per-packet overhead and per-byte costs. In some instances there was a zero per-byte cost. For example, the microengines share memory with the StrongARM, meaning there is no per-byte cost to transfer data. In this case, the per-packet cost captures the signaling overhead between the code running on the microengines and the code running on the StrongARM. Also note that in one instance we calculated a slight negative per-packet overhead. This is due to the fact that it is sometimes more efficient to handle small packets. Because all packets are at least 64 bytes long, in no case is the overall time for a packet ever negative.

As a simplification, we constrained the ports to be full-duplex and the services to have symmetric input and output bandwidth. These assumptions allow us to model the ports and microengines in aggregate. It also means that we can arbitrarily split the measured round-trip packet overhead cost between transmitting and receiving. (Every packet received by a processor will also be transmitted by that processor.) Tables 4.3 and 4.4 summarize the measured and derived parameters for this configuration. (See Appendix A.) For our experiments, the IRH contains the first 64-bytes of the packet plus an additional overhead of 4 bytes.

Parameter	Value	Units	Description
`pt_cps`	733	Mcyc/s	Pentium cycles/second
`pt_env_cpp`	1342	cyc/pkt	Pentium environment overhead.
`irh_oh`	4	bytes	IRH overhead.
`irh_ph`	64	bytes	IRH maximum payload.
`min_bpp`	64	bytes/pkt	Minimum packet size.
`max_bpp`	1518	bytes/pkt	Maximum packet size.
`port_spp`	1600	ns/pkt	Time required to send or receive a packet.
`port_spb`	80	ns/byte	

Table 4.3: Router Parameters for a Pentium PC. (See Appendix A for details.)

4.5.2 Workload Generation

Section 4.3 characterizes the service classes in a workload. What remains is the method of setting the number of services in a workload. On the one hand, if the services in a workload use too few resources for a given router, then any algorithm will succeed in placing all the services. On the other hand, if we have too many services in the workload (so that only a small fraction could possibly be placed), an offline algorithm will have an unfair advantage in that it can choose among many more configurations. The concern here is that if an offline algorithm is allowed to choose from a workload with a large number of

Parameter	Value	Units	Description
sa_cps	199.1	Mcyc/s	StrongARM cycles/second
ue_cpp	768	cyc/pkt	Microengine cycles/Packet.
sa2pt_dma_spp	3739	ns/pkt	Time on PCI bus when StrongARM sends to Pentium using DMA.
sa2pt_dma_spb	17.91	ns/byte	
sa2pt_pio_spp	3.27	ns/pkt	Time on PCI bus when StrongARM sends to Pentium using PIO.
sa2pt_pio_spb	40.30	ns/byte	
sa2pt_pio_max	166	bytes	Max packet size for StrongARM to use PIO on PCI.
pt2sa_spp	-34.94	ns/pkt	Time on PCI bus when Pentium sends to StrongARM using PIO.
pt2sa_spb	15.09	ns/byte	
sa_ue2_cpp	189	cyc/pkt	Cost to the StrongARM to receive from the microengines.
sa_ue2_cpb	0	cyc/byte	
sa_2ue_cpp	189	cyc/pkt	Cost to the StrongARM to send to the microengines.
sa_2ue_cpb	0	cyc/byte	
sa_pt2_cpp	186	cyc/pkt	Cost to the StrongARM to receive from the Pentium.
sa_pt2_cpb	0	cyc/byte	
sa_2pt_dma_cpp	186	cyc/pkt	Cost to the StrongARM to send to the Pentium using DMA.
sa_2pt_dma_cpb	0	cyc/byte	
sa_2pt_pio_cpp	27	cyc/pkt	Cost to the StrongARM to send to the Pentium using PIO.
sa_2pt_pio_cpb	8	cyc/byte	
pt_sa2_cpp	99	cyc/pkt	Cost to the Pentium to receive from the StrongARM.
pt_sa2_cpb	0	cyc/byte	
pt_2sa_cpp	99	cyc/pkt	Cost to the Pentium to send to the StrongARM.
pt_2sa_cpb	11	cyc/byte	
sa_env_cpp	513	cyc/pkt	StrongARM environment overhead.
NUMPORTS	8		Number of Ports

Table 4.4: Router Parameters for a Pentium PC with an IXP1200 EEB line card. (See Appendix A for details.)

services, the subset it selects may not be representative of the distribution of the workload as a whole. We want to make the workload large enough to bring out the differences in the online placement algorithms while keeping it small enough to make a reasonable comparison (for a given workload distribution) with offline placement algorithms.

Clearly, the appropriate size of the workload will depend on the capacity of the router. A “big” router will be able to handle a larger workload than a “small” router. For our experiments, we use the *minimum impact* of a service on a router as the basis for determining the size of the workload. To calculate the minimum impact (a utilization vector), we first calculate the service impacts, $\mathbf{i}_{s,p}$, of placing the service on each processor. The components of the minimum impact vector are calculated as the minimum value of the corresponding components of the service impacts.

To generate a workload, we randomly generate services attempting to match the desired distribution of **tiny**, **small**, **medium**, and **large** services. For each service we generate, we calculate its minimum impact. If the sum of the service's minimum impact and the running total of the minimum impacts (of the services already in the workload) is "acceptable," then the service is added to the workload; otherwise, the service is not added to the workload and the workload is complete. For our experiments, we add services to the workload until the total minimum impact port utilization and utilization of any switch or processor component would both be at least 1.

Table 4.5 lists the workloads for the IXP1200 EEB router along with the total minimum impact. There are several observations to make about this table. First, only the StrongARM, microengine, and port minimum impacts are shown. The Pentium and PCI bus minimum impacts are zero and are not shown. They are zero because the StrongARM has the capability of handling any of the services (in isolation) that the Pentium could handle. This means that for services that could be placed on either the StrongARM or the Pentium, the minimum impact on the Pentium (and PCI bus) is zero and the minimum impact on the StrongARM is smaller impact of placing the service on the StrongARM or placing the service on the Pentium (and using StrongARM cycles to move the packets).

Second, the first two workloads (3/51/45/1 and 5/49/45/1) have significantly more services (503 and 295, respectively) than the other workloads. This is due to the fact that they contain a large percentage of **small** services. The **small** services can be placed on any of the processors. (When placed on a microengine, they typically consume 30–

Target Distribution (percentages)	Actual Distribution (counts)				Total Services		Total Minimum Impact		
	tny	sml	med	lrg	N	σ	SA	UE	Port
3/51/45/1	14	256	227	6	503	40	0.88	0.97	4.32
5/49/45/1	14	145	133	3	295	32	0.52	0.96	2.65
7/47/45/1	13	91	88	2	194	16	0.34	0.97	1.56
9/45/45/1*	16	83	83	2	183	23	0.32	0.97	1.44
11/43/45/1	15	60	63	1	139	9	0.24	1.04	1.08
13/41/45/1	19	57	63	2	140	16	0.24	1.35	0.99
15/39/45/1	23	58	68	1	151	18	0.26	1.63	0.99
9/15/75/1	21	35	175	3	234	15	0.69	1.47	1.00
9/25/65/1	18	49	127	3	196	23	0.50	1.17	0.99
9/35/55/1	15	58	91	2	165	18	0.35	0.97	1.14
9/45/45/1*	16	83	83	2	183	23	0.32	0.97	1.44
9/55/35/1	16	100	64	2	182	32	0.25	0.95	1.57
9/65/25/1	13	99	39	2	153	16	0.16	0.96	1.60
9/75/15/1	14	126	26	2	168	12	0.10	0.96	1.99
9/45/33/13	14	72	53	21	160	20	0.22	0.97	1.27
9/45/35/11	14	69	54	17	154	12	0.21	0.94	1.27
9/45/37/9	14	71	59	14	158	27	0.23	0.93	1.31
9/45/39/7	14	72	62	11	159	12	0.24	0.95	1.19
9/45/41/5	14	73	67	8	163	18	0.26	0.95	1.35
9/45/43/3	14	74	70	5	164	21	0.28	0.99	1.22
9/45/45/1*	16	83	83	2	183	23	0.32	0.97	1.44

*Listed multiple times in the table.

Table 4.5: Workloads for the IXP1200 EEB router. The three horizontal sections correspond to the X-axes in the graphs of Figures 4.1, 4.2, and 4.3. Each row is the average across five generated workloads. The standard deviation of N is σ.

60% of the available budget.) Because they can be placed anywhere, the **small** services have a minimum impact of zero on the Pentium, StrongARM, and microengines. As a consequence, workloads with larger percentages of **small** services will tend to have more services overall.

Finally, because the **small**, **medium**, and **large** services always impact the ports, we can make some observations about the port column. Note that the port column (like the microengine column) is the aggregate utilization of all eight ports. For the three rows where the port impact is less than or equal to 1, we interpret this to mean that the router is not well suited to the corresponding distribution. That is, without attempting to optimally place the services, we already know that the microengines (in these cases), will be over utilized. This is due to the impact of the **tiny** services. By definition, these services must be placed on the statically scheduled microengines. From the table, we see that when there are 18 or more of these services, the microengines are oversubscribed.

4.5.3 Fixed Service Distribution

To establish a starting point, we evaluate the algorithms across five 9/45/45/1 workloads containing an average of 183 services. The results are reported in Table 4.6. Included in the table are the six possible Ordered algorithms along with the variants of the L2-Norm, Balance, Hybrid, and Scan algorithms. Early experiments showed that the standard versions of these algorithms (ending in a 1) did not perform as well as we had expected. Analyzing the service placement as well as the reason services which could not be placed led us to believe that the StrongARM was a bottleneck. By placing too many services on the StrongARM, we were preventing future services from being placed on the Pentium. We modified these algorithms to scale the StrongARM utilization by a multiplicative factor before computing the appropriate norm. The final digit in the algorithm name gives the scaling factor. By making the StrongARM artificially expensive, we were able to place more services on the router than we otherwise would have been able. Intuitively, this makes sense when one considers the topology of the router. The StrongARM cycles are a precious resource. When they are exhausted, services can neither be placed on the StrongARM nor can they be placed on the Pentium.

	Algorithm	Placement			Total Placed			Utilization				
		UE	SA	PT	N	σ	%*	Port	UE	SA	PCI	PT
Ordered	UE-PT-SA	90	0	85	174	22	99.2	1.44	0.99	0.33	0.48	0.41
	UE-SA-PT	90	85	0	174	22	99.2	1.44	0.99	0.74	0.00	0.00
	PT-UE-SA	36	0	90	127	15	72.1	1.09	0.97	1.00	0.64	0.58
	PT-SA-UE	36	0	90	127	15	72.1	1.09	0.97	1.00	0.64	0.58
	SA-UE-PT	36	88	3	127	18	72.1	1.08	0.97	1.00	0.01	0.01
	SA-PT-UE	36	88	3	127	18	72.1	1.08	0.97	1.00	0.01	0.01
L2-Norm	L2-Norm1	28	77	23	127	9	72.6	1.04	0.97	1.00	0.04	0.07
	L2-Norm2	25	69	35	129	9	73.3	1.04	0.97	1.00	0.06	0.11
	L2-Norm4	22	65	41	128	8	73.0	1.01	0.97	1.00	0.10	0.14
	L2-Norm8	22	55	59	136	6	77.2	1.07	0.97	1.00	0.32	0.28
Balanced	Balance1	28	65	46	138	18	78.7	1.06	0.97	1.00	0.25	0.22
	Balance2	64	36	76	176	22	**100.0**	1.44	0.99	0.76	0.44	0.37
	Balance4	79	13	82	175	22	99.5	1.44	0.99	0.49	0.47	0.40
	Balance8	86	5	84	175	22	99.4	1.44	0.99	0.38	0.47	0.40
Hybrid	UE;S-P-1	90	46	39	174	22	99.2	1.44	0.99	0.52	0.06	0.12
	UE;S-P-2	90	30	55	174	22	99.2	1.44	0.99	0.45	0.09	0.17
	UE;S-P-4	90	22	63	174	22	99.2	1.44	0.99	0.42	0.15	0.21
	UE;S-P-8	90	2	83	174	22	99.2	1.44	0.99	0.33	0.45	0.39
Scan	Scan1	16	57	82	156	13	88.7	0.84	0.94	0.99	0.44	0.38
	Scan2	16	56	84	156	12	89.1	0.85	0.94	0.99	0.47	0.40
	Scan4	16	56	84	156	12	89.1	0.85	0.94	0.99	0.47	0.40
	Scan8	16	56	84	156	12	89.1	0.85	0.94	0.99	0.47	0.40

*Normalized to the best algorithm for this workload.

Table 4.6: Algorithm performance on the IXP1200 EEB router across five 9/45/45/1 workloads containing an average of 183 services. Due to round-off error, the sum of the placements does not always total N. The standard deviation of N is σ.

There are several observations to make about these results. First, most rows have a port utilization that exceeds 1. This is because the algorithms are using a provisioning vector that has a port component that is infinite (the other components are set to 1). This allows the algorithms to place more services and not be limited by the ports. When the port utilization is greater than 1, it means that the processors and switches in the router were not the limiting factor for the workload. Note that we are not explicitly modeling the IX bus in this version of the simulator; however, because we have measured the IX bus and microengines operating at 3.47 Mpps, we know that they could handle 23.3 times the traffic of a single 100 Mbps Ethernet port. This is 2.9 times the traffic of the eight ports in our router. As long as the port utilization remains under 2.9, the results are still valid.

Second, the Ordered algorithms, not surprisingly, are sensitive to the order selected, with results ranging from 72.1% to 99.2%. While it is clear that both orders that prefer the StrongARM are not likely to yield good results, the relative behavior of the other orders is not so obvious. We expect that with more complex architectures, the correct order will be even harder to predict. Thus, while intuition and simulations might suggest a particular order, it is difficult to draw any general conclusions about its behavior.

Third, exploiting knowledge about the vulnerable components in the system (the StrongARM) had a noticeable effect on the Balanced algorithm (increasing the success rate from 78.7% to 100%). While there was no net effect on the Hybrid algorithms, we observed an expected downward trend on the number of services placed on the StrongARM as its weight was increased. While the Ordered `UE-PT-SA` and the Hybrid algorithms placed the same total number of services (174), the placement distribution was different; as the cost of the StrongARM increases, the Hybrid algorithm behaves more and more like the `UE-PT-SA` algorithm. The intuitive difference is that the Ordered `UE-PT-SA` algorithm blindly places services on the Pentium (after considering the microengines) and the Hybrid algorithm makes an informed decision. It does not bother to put a service on the Pentium which would cause the StrongARM to expend more effort forwarding the packets than it would by performing the service itself.

Fourth, the offline algorithms performed worse than the best online algorithms. Recall that the services were added in the order of their impact on the system, with the lesser impact services selected first. It turns out that the smallest services (those that would probably fit on the microengines) have a high impact due to their high packet rates, and so they were added last. This is confirmed by the fact that only 16 services were loaded onto the microengines for these algorithms. We also tried running the Scan algorithm in the reverse direction, with the highest impact services added first. This makes a certain amount of sense because it allows you to place the worst-case services onto a highly loaded system, with the best-base services filling in the "cracks." From these observations, we conclude that the random nature of the workload works in favor of the online algorithms.

Finally, the Balance and Hybrid algorithms outperformed the L^2 norm algorithms. This is because both the Balance and Hybrid algorithms use the L^∞ norm, which better models the shape of the region defined by the provisioning vector.

4.5.4 Varying Service Distributions

We next consider the behavior of the algorithms under different workload distributions. In the graphs which follow, we exclude the `SA-PT-UE` algorithm because its behavior was nearly identical to the `SA-UE-PT` algorithm. In addition, we only show the one norm-based algorithm of each type that had the best performance for the 9/45/45/1 workload.

We vary the service distributions one pair at at time (tiny and small, small and medium, medium and large). For each graph, the fraction placed is relative to the best performing algorithm for the given workload. Therefore, for each workload, there will be an algorithm with a relative fraction placed value of 1.

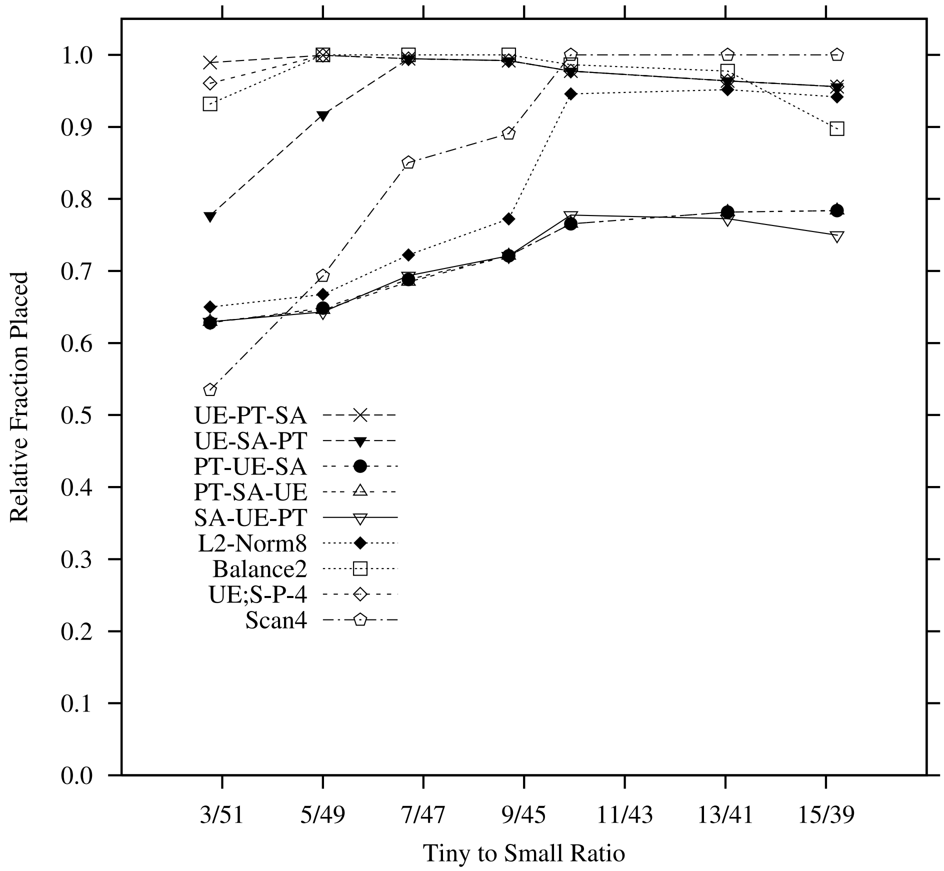

Figure 4.1: Relative algorithm placement performance under different non-port-limited workloads. The medium and large fraction of the workloads are fixed at 45% and 1% respectively

Figure 4.1 shows the consequence of varying the ratio of tiny to small services, where the former are applied to all packets and the later are applied on a per-flow basis. As we can see, all the algorithms have a slight trend upward as more tiny services are included. This may be due to the fact that smaller services pack better. Moreover, the results show that both UE-PT-SA and UE;S-P-4 have consistently good behavior across this set of workloads. This leads us to conclude that placing services on the microengines first is a good idea. Similarly, both PT-SA-UE and SA-UE-PT have similar, poor behavior. This re-enforces the insight that placing services on the StrongARM before attempting to place them on the microengines is a bad idea.

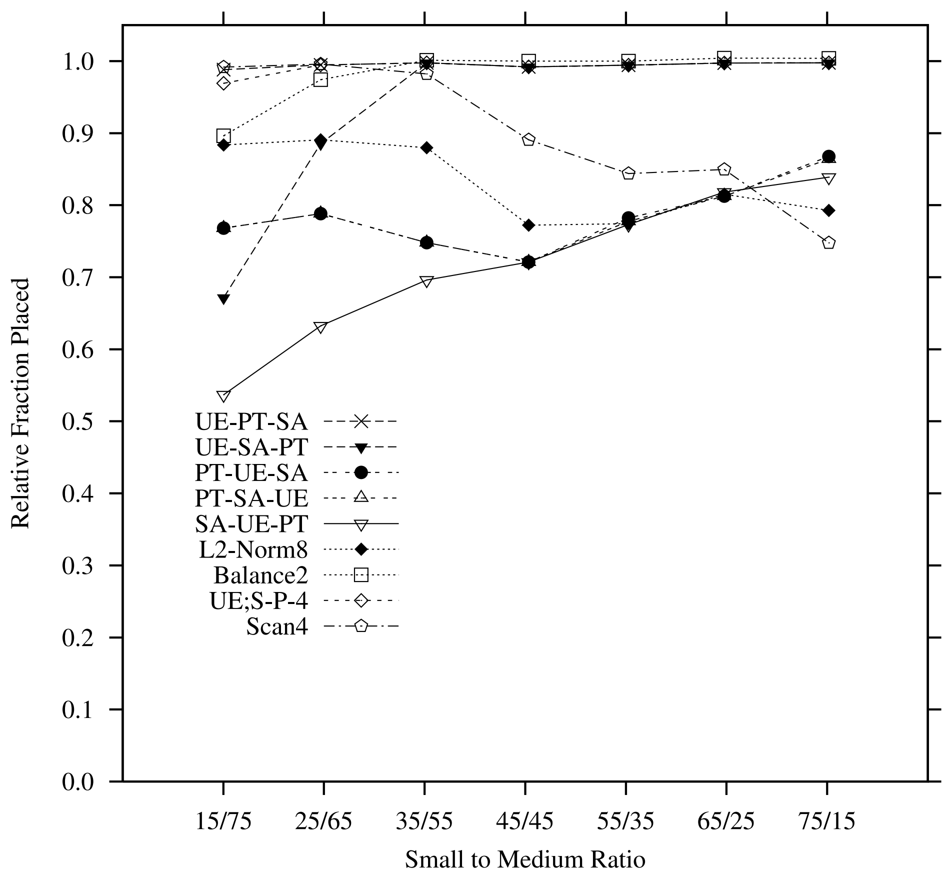

Figure 4.2: Relative algorithm placement performance under different non-port-limited workloads. The tiny and large fraction of the workloads are fixed at 9% and 1% respectively

Next, Figure 4.2 shows the consequence of varying the ratio of small to medium services. As in Figure 4.1, the general trend of the better performing algorithms is slightly upward. Interestingly, the offline Scan4 algorithm had the opposite trend.

A modified version of `Scan4` where the services with the largest impact were placed first had an opposite trend and did perform better for 55/35, 65/25, and 75/15 (reaching nearly 1.0). However, this reversed version performed significantly worse under all the other workloads we measured.

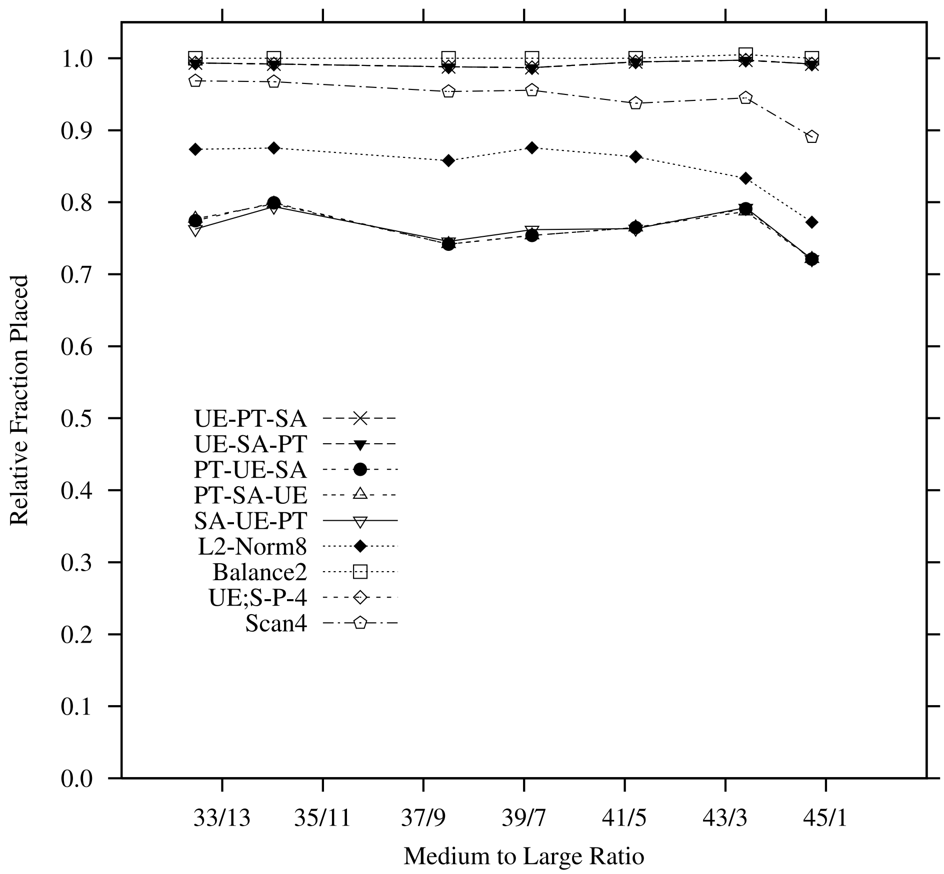

Figure 4.3: Relative algorithm placement performance under different non-port-limited workloads. The tiny and small fraction of the workloads are fixed at 9% and 45% respectively

Finally, Figure 4.3 plots the behavior of the algorithms as ratio of medium to large services changes. The graph is relatively flat for most of the well-performing algorithms, meaning that most are insensitive to the medium-to-large mix.

4.6 Experimental Results on PMC694

This section explores an alternative hardware implementation based on the PMC694. While the IXP1200 EEB router is a three-tier processor hierarchy, the PMC694 router is only a two-tier processor hierarchy. Another significant difference is that the IXP1200 EEB architecture assigns a microengine context to each port while the PowerPC in the PMC694 architecture must service multiple (two in this case) ports.

4.6.1 PMC694 Router Parameters

We have evaluated the placement algorithms using a hardware configuration consisting of a 733 MHz Pentium III PC with a 32 bit, 33 MHz PCI bus connected to an off-the-shelf 266 MHz PMC694 intelligent line card with two 100 Mbps Ethernet ports.

As in the IXP1200 router, we constrained the ports to be full-duplex and the services to have symmetric input and output bandwidth. Tables 4.3 and 4.7 summarize the measured and derived parameters for this configuration. (See Appendix A.) Because the PowerPC is now at the bottom of the hierarchy, we must statically schedule a portion of the available processor cycles so that we can support both the initial classification (environment) and the **tiny** services (applied to every packet) at line speed. The **tiny** services must fit within a worst case budget of 657 cpp. This is 893 cpp (from Table 3.4) less the environment cost of 236 cpp (from Table 4.7). In addition, the overall expected cycle cost must fit within a total budget of 297.6 Kpps $\times$ 657 cpp, or 195.5 Mcps. (Note that this is 74% of the available 266 Mcps on the PowerPC.)

Parameter	Value	Units	Description
pp_cpp	893	cyc/pkt	PowerPC cycles/packet
pp2pt_spp	-581.1	ns/pkt	Time on PCI buses when PowerPC sends to Pentium using DMA.
pp2pt_spb	38.34	ns/byte	
pt2pp_spp	-33.23	ns/pkt	Time on PCI buses when Pentium sends to PowerPC using PIO.
pt2pp_spb	15.08	ns/byte	
mc2pp_spp	-33.23	ns/pkt	Time on secondary PCI bus when MAC sends to PowerPC.
mc2pp_spb	15.08	ns/byte	
pp2mc_spp	-581.1	ns/pkt	Time on secondary PCI bus when PowerPC sends to MAC.
pp2mc_spb	38.34	ns/byte	
mp_ofact	0.7		Secondary PCI bus occupancy factor.
pp_mc2_cpp	0	cyc/pkt	Cost to the PowerPC to receive from the ports.
pp_mc2_cpb	0	cyc/byte	
pp_2mc_cpp	0	cyc/pkt	Cost to the PowerPC to send to the ports.
pp_2mc_cpb	0	cyc/byte	
pp_pt2_cpp	186	cyc/pkt	Cost to the PowerPC to receive from the Pentium.
pp_pt2_cpb	0	cyc/byte	
pp_2pt_cpp	186	cyc/pkt	Cost to the PowerPC to send to the Pentium.
pp_2pt_cpb	0	cyc/byte	
pt_pp2_cpp	99	cyc/pkt	Cost to the Pentium to receive from the PowerPC.
pt_pp2_cpb	0	cyc/byte	
pt_2pp_cpp	99	cyc/pkt	Cost to the Pentium to send to the PowerPC.
pt_2pp_cpb	11	cyc/byte	
pp_env_cpp	236	cyc/pkt	PowerPC environment overhead.
NUMPORTS	2		Number of Ports

Table 4.7: Router Parameters for a Pentium PC with an PMC694 line card. (See Appendix A for details.)

4.6.2 Workload Generation

Table 4.8 lists the workloads for the PMC694 router along with the total minimum impact. There are a couple of observations to make about this table. First, only the PowerPC, secondary PCI bus, and port minimum impacts are shown. The Pentium and primary PCI bus minimum impacts are zero and are not shown. They are zero because the PowerPC has the capability of handling any of the services (in isolation) that the Pentium could handle. This means that for services that could be placed on either the PowerPC or the Pentium, the minimum impact on the Pentium (and primary PCI bus) is zero and the minimum impact on the PowerPC is smaller impact of placing the service on the PowerPC or placing the service on the Pentium (and using PowerPC cycles to move the packets).

Target Distribution (percentages)	Actual Distribution (counts)				Total Services		Total Minimum Impact		
	tny	sml	med	lrg	N	σ	PP	sPCI	Port
3/51/45/1	4	66	59	2	130	8	0.41	0.99	1.12
5/49/45/1	6	62	57	2	127	12	0.64	0.99	1.12
7/47/45/1	9	62	60	2	133	3	0.86	0.97	1.09
9/45/45/1*	12	61	61	1	136	8	0.94	0.95	1.07
11/43/45/1	15	56	58	1	130	9	1.25	0.88	0.99
13/41/45/1	19	57	63	2	140	16	1.64	0.87	0.99
15/39/45/1	22	57	67	1	148	20	1.93	0.88	0.99
9/15/75/1	21	35	175	3	234	15	1.80	0.89	1.00
9/25/65/1	18	49	127	3	196	23	1.46	0.88	0.99
9/35/55/1	13	51	80	2	146	12	1.12	0.89	1.00
9/45/45/1*	12	61	61	1	136	8	0.94	0.95	1.07
9/55/35/1	11	67	44	2	124	9	0.92	0.93	1.05
9/65/25/1	9	65	25	1	101	3	0.94	0.95	1.06
9/75/15/1	8	69	14	1	92	7	0.81	0.98	1.11
9/45/33/13	12	60	45	18	134	8	1.02	0.94	1.06
9/45/35/11	12	60	46	14	132	18	1.05	0.95	1.06
9/45/37/9	12	61	50	12	136	5	1.01	0.92	1.04
9/45/39/7	13	64	56	10	143	6	1.12	0.92	1.04
9/45/41/5	12	62	56	7	137	15	1.07	0.94	1.06
9/45/43/3	12	61	59	4	136	3	1.12	0.87	0.98
9/45/45/1*	12	61	61	1	136	8	0.94	0.95	1.07

*Listed multiple times in the table.

Table 4.8: Workloads for the PMC694 router. Each row is the average across five generated workloads. The standard deviation of N is σ.

Second, as in the IXP1200 EEB case, we can make some observations about the router based on the port column. Note that the port column is the aggregate utilization of both ports. Where the port impact is less than or equal to 1, we interpret this to mean that the router is not well suited to the corresponding workload. It is interesting to note that the all four workloads not well suited to the IXP1200 EEB were also not well suited to the PMC694.

4.6.3 Fixed Service Distribution

Table 4.9 shows the algorithm performance on the PMC694 across five 9/45/45/1 workloads containing an average of 136 services. This is analogous to Table 4.6 for the IXP1200 EEB. Because there are only two processors in the hierarchy, there are only two Ordered algorithms and the Hybrid algorithms degenerate into one of these two Ordered algorithms. To make a better comparison with the IXP1200 EEB results, we also included variants (ending in 2, 4, or 8) where the PowerPC (like the StrongARM) was made more expensive by the corresponding scaling factor.

There are several observations to make about these results. First, neither Ordered algorithm performed better than the other algorithms. Placing too many services on the PowerPC (as in the `PP-PT` case) caused the PowerPC to saturate; placing too many services on the Pentium (as in the `PT-PP` case) caused the secondary PCI bus to saturate. In fact, in the case of the `PT-PP` algorithm, the secondary PCI bus saturated before the ports.

Second, for the L2-Norm, Balance, and Scan algorithms, the number of services placed on the PowerPC drops as the scaling factor increases. This is an expected result. Interestingly, the scaling factor associated with the PowerPC (analogous to the scaling

	Algorithm	Placement		Total Placed			Utilization				
		PP	PT	N	σ	%*	Port	sPCI	PP	pPCI	PT
Ord.	`PP-PT`	130	1	131	12	96.6	1.02	0.91	0.96	0.00	0.00
	`PT-PP`	12	114	126	6	93.1	0.97	1.00	0.92	0.20	0.19
L2-Norm	`L2-Norm1`	126	7	133	10	98.2	1.05	0.93	0.95	0.00	0.01
	`L2-Norm2`	97	39	136	8	100.0	1.06	0.95	0.94	0.01	0.03
	`L2-Norm4`	91	45	135	8	99.9	1.06	0.95	0.94	0.02	0.04
	`L2-Norm8`	90	46	135	8	99.9	1.06	0.96	0.94	0.02	0.04
Balanced	`Balance1`	107	29	136	8	100.0	1.05	0.97	0.94	0.05	0.04
	`Balance2`	67	64	131	7	96.8	1.02	0.98	0.93	0.11	0.09
	`Balance4`	63	69	133	6	97.8	1.02	0.99	0.92	0.12	0.09
	`Balance8`	63	70	133	6	97.8	1.02	0.99	0.92	0.12	0.09
Scan	`Scan1`	117	18	136	8	100.0	1.06	0.94	0.91	0.01	0.01
	`Scan2`	108	27	136	8	100.0	1.06	0.95	0.90	0.01	0.02
	`Scan4`	83	52	135	8	99.9	1.04	0.95	0.90	0.05	0.05
	`Scan8`	65	70	135	7	99.4	1.01	0.98	0.91	0.12	0.09

*Normalized to the best algorithm for this workload.

Table 4.9: Algorithm performance on the PMC694 router across five 9/45/45/1 workloads containing an average of 136 services. Due to round-off error, the sum of the placements does not always total N. The standard deviation of N is σ.

factor for the StrongARM) had a relatively small net effect. This may be due to the fact that the PowerPC on the PMC694 must perform the tasks that both the microengines and the StrongARM on the IXP1200 EEB perform.

Third, compared to the IXP1200 EEB, the algorithms for the PMC694 tend to place a larger fraction of the services in the workloads. This is due to the fact that the PMC694 architecture is simpler than the IXP1200 EEB (i.e., fewer levels in the processor hierarchy). As the architecture gets simpler, our method of generating workloads emulates the behavior of the placement algorithms. For example, consider a router with a single processor (i.e., a PC with “dumb” network interface cards). Because there is only one place to put a service (on the Pentium), the minimum impact used to generate the workload will be identical to the actual impact. As a result, our method of generating workloads—adding services until no more will fit—implies that the resulting workload will always fit. Another consequence of a simpler architecture is that the algorithms have fewer choices available and will, naturally, yield similar results. In the example case of a single processor, all algorithms degenerate into the trivial "place it on the only processor" algorithm.

Finally, compared to the IXP1200 EEB, the port utilization tends to be closer to 1. This can be interpreted in two ways. On the one hand, it means that the PMC694 router has less headroom—that is, less excess capacity. On the other hand, it means that the PMC694 router is better optimized.

4.7 Evaluation

This chapter models an extensible router as a hierarchy of processor and switching elements, and studies a collection of algorithms for placing the services onto this architecture. As expected, the algorithms have a noticeable effect on the number of services that can be placed, with a 39% difference between the best and worst algorithm we evaluated. In addition to demonstrating feasible algorithms for resource allocation and admission control, we offer the following observations.

First, including architecture-specific knowledge is important to the performance of the algorithms. For example, weighting the impact of placing services on the StrongARM improves both the L2-Norm and L^∞ norm (Balance) algorithms, and trying to first place services on the microengines (resulting in the Hybrid algorithm) yields additional improvements. In general, the Hybrid algorithm's combination of the L^∞ norm strategy and architecture-specific knowledge performed best across the workloads we considered. The L^∞ norm strategy also yielded consistently good results on the PMC694.

Second, the best-performing algorithms—Balance, Hybrid, and `UE-PT-SA`—were largely insensitive to the workload mix, performing equally well as we varied the distribution of service processing times. Generally, all of the algorithms performed better when presented with a larger ratio of smaller-sized jobs, which is not surprising since smaller jobs offer more opportunities for placement.

Third, some of the simple Ordered algorithms on the IXP1200 EEB performed surprisingly well. We believe these algorithms should be viewed as an approximation to the Hybrid algorithm, which like the best case order algorithm (`UE-PT-SA`), first tries to fit the service on the microengines. Although our particular configuration did not illustrate this advantage, we believe the Hybrid algorithm should scale better than `UE-PT-SA` and that Balance should scale better than Hybrid for more complex combinations of processors above the microengine level. The Ordered algorithms on the PMC694 were inferior (either placed fewer services or had a higher variance) to the norm-based algorithms. Together, this tells us that, in general, the norm-based algorithms outperform the Ordered algorithms.

Finally, the evaluation of the algorithms exposed three interesting observations about the hardware. One is that even though the StrongARM is on the critical path to the Pentium, originally leading us to believe that placing any services on the StrongARM was a bad idea, we found that the StrongARM was able to host a significant fraction of the workload, approximately 20% for the best algorithms. A second observation is that while there might be a market for the IXP1200 as a stand-along system, adding the Pentium to the hierarchy has significant benefit at relatively little cost. One ramification of adding the Pentium is that it can be the processor that runs a general-purpose OS (e.g., Linux) in support of legacy services (e.g., BGP), leaving the StrongARM free to run a minimal runtime system that is sufficient for the intermediate-sized services for which it seems well suited. The last observation is that despite concerns about the primary PCI bus being the bottleneck, it was never the first saturated device in our experiments; generally the StrongARM was the bottleneck.

4.7.1 Extensibility and Robustness

Recall from Chapter 1 that our definition of robustness consists of two parts: routers (1) must be able to read and classify packets at line speed and (2) must honor the processing guarantees they make. At the end of the previous chapter, we concluded that our implementation is able to read and classify packets at line speed. In this chapter we argue that the router can make processing guarantees, in the form of admission control, that it can honor. We show this by demonstrating a method of assigning services to processors in a way that we do not oversubscribe the critical resources of the extensible router.

Because our algorithms do admit services when there are resources available, we conclude that our resource allocation mechanism provides extensibility. In addition, because our algorithms successfully prevent the router from over-committing its resources, we conclude that our resource allocation mechanism provides robustness.

Chapter 5

Conclusions

This dissertation demonstrates that one can to build a router that is simultaneously extensible and robust using commercially available, PC-based components. We show this by presenting a novel architecture, key implementation methods for PC-based systems, performance evaluations using two different line cards, and several resource allocation algorithms.

5.1 Research Contribution

The main contribution of this thesis is to demonstrate that we can build a router from commercially available, PC-based components (specifically, multi-port, programmable line cards) that is simultaneously extensible and robust. To show this, we (1) describe an architecture, called VERA, that is extensible and robust; (2) present implementation techniques to realize this architecture on a PC-based router; and (3) characterize and analyze the problem of mapping the functions implementing the router services to the various, heterogeneous processors comprising the router in a way to preserve robustness.

The architectural contribution is to identify the design decisions we made while defining and implementing VERA. The design consists of three layers. First, VERA defines an upper interface, called the router abstraction, that has explicit support for extensibility. The router abstraction consists of three elements: classifiers, forwarders, and schedulers. In addition, we support a classification hierarchy. Second, VERA defines a lower interface, called the hardware abstraction, that hides the details of the underlying hardware. The hardware abstraction also consists of three elements: processors, ports, and switches. The processors within the router are organized into a processor hierarchy. Our architecture defines the mapping of an abstract forwarding path (through the elements of the router abstraction) onto a switching path (through the elements of the hardware abstract) as a concrete forwarding path. Third, a distributed OS ties the upper and lower abstractions together.

During the implementation of key elements of VERA, we have uncovered the many details involved in data movement and queue management in a distributed environment (distributed queues). We have determined that a key design consideration for our router architecture is that the design must support the independent impact of flows. To support independent impact and ensure that our router is robust (can handle packets at line rate), we have introduced a combination of static and dynamic scheduling of processor schedules.

Using performance measurements on actual hardware, this thesis characterizes and analyzes the problem of mapping the functions implementing the router services to the various, heterogeneous processors comprising the router hardware. This includes defining a hardware performance model. Our analysis is based on a utilization vector representing the overall utilization of the router. We have found that an algorithm that minimizes the L^{∞} norm of the utilization vector will be able to place more services on the router than algorithms which do not consider the utilization or algorithms that use a Euclidean norm.

All of these contributions are in the context of specific hardware configurations consisting of both a three-level and a two-level processor hierarchy. We demonstrate our two-level design on a Pentium augmented with a PowerPC-based line card and our three-level design on a Pentium augmented with an IXP1200 network processor based line card. While we have focused on two, relatively small, router configurations, we believe our basic architecture applies equally well to richer configurations. In general, as network processors become more prevalent in high-end routers, we expect our techniques to also apply there as well. In the end, we expect the distinction between “hardware-based” and “software-based” routers to become less meaningful.

5.2 Future Work

There are several directions we can extend the work presented in this thesis. First, as part of an ongoing effort to incorporate VERA into SILK [3, 54], we can continue to develop the software base. Specifically, we would like to update the control interface to the VERA Linux device driver so that SILK paths can control services installed on the processors on the line cards.

Second, we would like to explore the design space of VERA routers. One direction is to build a larger system consisting of multiple PCs. By using multiple PCs and labeling one as the root processor we will have a deeper processor hierarchy. This will let us explore the space toward systems like Suez [8, 50] and SHRIMP [5]. Multi-PC systems will allow us to validate or modify the distributed queue technique across LAN/SAN links. Another area of the design space is to explore multiple line cards (possibly of different types) on the PCI bus. This will let us better understand the point at which the PCI bus may (or may not) become the bottleneck.

Third, we would like to enhance the resource allocation model presented in Chapter 4. Specifically, we would like to include memory in the utilization vector. This will allow us, for example, to limit the number and/or size of the services installed on a processor. This is particularly relevant to the microengines which only have a 1024 word instruction store. In addition to extending the model, we would like to enhance its fidelity by including more measurements of the hardware and fewer parameters derived indirectly from other measurements. An important part of validating our router model is developing real-world workloads.

Finally, we would like to explore the possibility designing and building a *semaphore server* for the PCI bus. This would be a simple hardware device for the PCI bus that would allow multiple bus masters to acquire and release semaphores (protecting shared queues and buffers) without spinning on shared memory locations across the bus. This would act as an extension (or replacement) to the I_2O hardware registers we exploit in our current implementation. By having this functionality on a separate board, we no longer would require I_2O support on the line cards. Advanced capabilities might include support for a hardware callback—bus masters could register their request for a semaphore. When it is released, the semaphore server could directly contact the queued bus master.

Appendix A

Router Parameter Table Calculations

This appendix gives the calculations for the entries in Tables 4.3, 4.4, and 4.7. The parameter names use the following abbreviations:

`pt` Pentium

`sa` StrongARM

`ue` Microengine

`pp` PowerPC

`mc` (MAC) port on the PMC694

`bpp` bytes/packet

`cpb` cycles/byte

`cpp` cycles/packet

`cps` cycles/second

`spb` seconds/byte

`spp` seconds/packet

A.1 Common Router Parameters

This section gives the calculations for the entries in Tables 4.3.

pt_cps

Pentium III cycles per second. Set to 733 Mcps.

pt_env_cpp

Pentium III *environment cost* in cycles/packet. The environment cost is the overhead cost per packet to classify and shuffle the packet within the CPU. It is the total number of cycles per packet less the send/receive cost and less the forwarding function cost. Peterson, *et al.* [47] reports that a 450 Mcps (MHz) Pentium II can forward 290.0 Kpps through "IP Fast Path" code (IP--). Dividing these two values gives a total of 1550 cpp. Their code used a DMA-based Ethernet chip and a polling device driver. Because their code performed a differential checksum computation (rather than recalculating the checksum from scratch), only a few bytes of the packet are accessed. As a result, there is no per-byte cost in this code. On a Pentium III, we measured the IP-- function (decrement TTL, update checksum) at 10 cpp. We assume that the cycles/instruction (CPI) for the Pentium II and Pentium III are approximately the same.[1] The total send/receive cost is the sum of `pt_sa2_cpp` and `pt_2sa_cpp` (see below), or 198 cpp. We conclude that the Pentium III environment cost (`pt_env_cpp`) is $1550 - 10 - 198$, or 1342 cpp.

irh_oh, irh_ph

The internal routing header (IRH) overhead, `irh_oh`, is set to 4 bytes. This is enough to tag a block. The second parameter, `irh_ph`, indicates the maximum

[1] An informal search of the Internet indicates that this is the conventional wisdom.

amount of data from the packet that is sent in the IRH. We set this to 64 bytes so that the smallest packet will fit in an IRH.

`min_bpp`, `max_bpp`

The minimum and maximum Ethernet packets sizes in bytes/packet. For our model, we do not include the preamble or the SFD (Start Frame Delimiter) fields in the packet size range. The sizes are derived from [25] and include the destination address (6 octets), source address (6 octets), length/type field (2 octets), the data (46 to 1500 octets), and the FCS (frame check sequence) (4 octets).

`port_spp`, `port_spb`

Time on the wire for a 100 BASE–T (100 Mbps) Ethernet packet. The overhead time per packet (`port_spp`) is the interFrameGap of 960 ns plus the time to transmit the preamble (7 octets = 56 bits) and the SFD (1 octet = 8 bits) at 10 ns/bit, or 1600 ns. The per-byte time (`port_spb`) is 80 ns (8 bits per octet at 10 ns per bit).

A.2 IXP1200 EEB Router Parameters

This section gives the calculations for the entries in Table 4.4.

`sa_cps`

StrongARM cycles per second. Set to 199.1 Mcps.

`ue_cpp`

The statically scheduled microengines use a fixed per packet budget. Our value is calculated based on the VRP of Section 3.3.3 as $240\,\text{cyc/pkt} + 24\,\text{accesses/pkt} \times 22\,\text{cyc/access}$, or 768 cpp. Note that rather than modeling processing cycles and memory access separately, we convert the memory accesses to processing cycles at a rate of 24 accesses/pkt. In a deployed system, we must ensure that there are no more than 22 memory accesses in the VRP code for each packet.

sa2pt_dma_spp, sa2pt_dma_spb

Time on the PCI bus when the StrongARM sends to the Pentium using DMA. From Table 3.2, we have the data points (64 bytes, 13.10 MByte/sec) and (1500 bytes, 49.01 MByte/sec). Converting the rates to time gives the data points (64 bytes, 4.885 μs) and (1500 bytes, 30.61 μs). A linear fit gives an overhead (`sa2pt_dma_spp`) of 3739 ns/pkt and a per-byte time (`sa2pt_dma_spb`) of 17.91 ns/byte.

sa2pt_pio_spp, sa2pt_pio_spb

Time on the PCI bus when the StrongARM sends to the Pentium using PIO. From Table 3.2, we have the data points (64 bytes, 24.78 MByte/sec) and (1500 bytes, 24.81 MByte/sec). Converting the rates to time gives the data points (64 bytes, 2.583 μs) and (1500 bytes, 60.46 μs). A linear fit gives an overhead (`sa2pt_pio_spp`) of 3.272 ns/pkt and a per-byte time (`sa2pt_pio_spb`) of 40.30 ns/byte.

sa2pt_pio_max

Since the StrongARM DMA overhead is so great, small packets should be sent using PIO. The intersection of the PIO and DMA lines occurs at 166.8 bpp so we set `sa2pt_pio_max` to 166.

pt2sa_spp, pt2sa_spb

Time on the PCI bus when the Pentium sends to the StrongARM using PIO. From Table 3.2, we have the data points (64 bytes, 68.73 MByte/sec) and (1500 bytes, 66.35 MByte/sec). Converting the rates to time gives the data points (64 bytes, 931.2 ns) and (1500 bytes, 22.61 μs). A linear fit gives an overhead (`pt2sa_` spp) of −34.84 ns/pkt and a per-byte time (pt2sa_spb) of 15.09 ns/byte. The negative overhead is due to the fact that the Pentium is slightly more efficient at sending 64-byte blocks than sending 1500-byte blocks.

sa_ue2_cpp, sa_ue2_cpb, sa_2ue_cpp, sa_2ue_cpb

The StrongARM cost to receive from (ue2) and send to (2ue) the microengines. The per byte cost (cpb) is zero because the microengines share packet memory with the StrongARM. Dividing the microengine to StrongARM to microengine rate of 526 Kpps with a null forwarder (see Section 3.3.2) into the 199.1 Mcps StrongARM processor rate, we get a round trip cost of 378 cpp. We arbitrarily assign half of the cost (189 cpp) to each leg of the trip. (This division only becomes important when we introduce forwarders which have a significant imbalance in the number of packets in versus the number of packets out. This is not the case for our current simulations.)

sa_pt2_cpp, sa_pt2_cpb, sa_2pt_dma_cpp, sa_2pt_dma_cpb

The StrongARM cost to receive from (pt2) and send (using DMA) to (2pt) the Pentium. The per byte receive cost (sa_pt2_cpb) is zero because the Pentium writes the packet data directly into the StrongARM memory across the PCI bus. The per byte send cost (sa_2pt_dma_cpb) is zero because the DMA engine moves the data, not the StrongARM. Dividing the StrongARM to Pentium to StrongARM rate of 534 Kpps with a null forwarder (see Section 3.3.2) into the 199.1 Mcps StrongARM processor rate, we get a round trip, per packet, cost of 372 cpp. Again, we arbitrarily assign half of the cost (186 cpp) to each leg of the trip.

sa_2pt_pio_cpp, sa_2pt_pio_cpb

The StrongARM cost to send (using PIO) to the Pentium. Using the per byte time on the PCI bus (`sa2pt_pio_spb`) of 40.30 ns and a 199.1 Mcps clock rate gives a per byte cost (`sa_2pt_pio_cpb`) of 8 cpb. We model the StrongARM per packet cost as taking the same time as the Pentium per packet cost (`pt_2sa_cpp`) of 99 cpp and scale it based on the ratio of the CPU clock rates to give a per packet cost (`sa_2pt_pio_cpp`) of 27 cpp.

pt_sa2_cpp, pt_sa2_cpb, pt_2sa_cpp, pt_2sa_cpb

The Pentium cost to receive from (`sa2`) and send to (`2sa`) the StrongARM. The per byte receive cost (`pt_sa2_cpb`) is zero because the StrongARM writes the packet data directly into the Pentium memory across the PCI bus. The remaining Pentium costs are derived from Table 3.3. We expect the behavior to be linear and model it as a fixed overhead, P, and a per byte cost, B, using the following simultaneous equations:

$$\begin{aligned} P + (64\,\text{byte}) * B + 500\,\text{cpp} &= 733\,\text{Mcps}/534\,\text{Kpps} \\ P + (1500\,\text{byte}) * B + 800\,\text{cpp} &= 733\,\text{Mcps}/43.6\,\text{Kpps} \end{aligned}$$

Solving gives $P = 198$ cpp (round-trip) and $B = 11$ cpb (send only). Again, we equally split the round-trip overhead between sending (`pt_sa2_cpp`) and receiving (`pt_2sa_cpp`). The per byte send cost (`pt_2sa_cpb`) is 11 cpb.

sa_env_cpp

StrongARM environment cost in cycles/packet. We assume that, except for classification, the environment overhead on the StrongARM will be 25% of that the Pentium. From [56] we assume that prefix-match classification requires on average

236 cycles per packet. For our model, we set the StrongARM environment cost to be $0.25(1342-236)+236$, or 513 cpp.

NUMPORTS

The number of 100 Mbps ports on the IXP1200 EEB. Set to 8.

A.3 PMC694 Router Parameters

This section gives the calculations for the entries in Table 4.7.

pp_cpp

Available PowerPC cycles per packet for minimum sized packets arriving at the maximum rate across all ports. From Table 3.4 (page 86).

pp2pt_spp, pp2pt_spb

Time on the PCI buses when the PowerPC sends to the Pentium using DMA. From Table 3.1, we have the data points (64 bytes, 34.19 MByte/sec) and (1500 bytes, 26.35 MByte/sec). Converting the rates to time gives the data points (64 bytes, 1872 ns) and (1500 bytes, 56930 ns). A linear fit gives an overhead (`pp2pt_spp`) of -581.1 ns/pkt and a per-byte time (`pp2pt_spb`) of 38.34 ns/byte. The negative overhead is due to the fact that the PowerPC is more efficient at sending 64-byte blocks than sending 1500-byte blocks.

pt2pp_spp, pt2pp_spb

Time on the PCI buses when the Pentium sends to the PowerPC using PIO. From Table 3.1, we have the data points (64 bytes, 68.70 MByte/sec) and (1500 bytes, 66.44 MByte/sec). Converting the rates to time gives the data points (64 bytes, 931.6 ns) and (1500 bytes, 22580 ns). A linear fit gives an overhead (`pt2pp_` spp) of $-33.23\,\mathrm{ns/pkt}$ and a per-byte time (pt2pp_spb) of $15.08\,\mathrm{ns/byte}$. The negative overhead is due to the fact that the Pentium is more efficient at sending 64-byte blocks than sending 1500-byte blocks.

mc2pp_spp, mc2pp_spb

Time on the secondary PCI bus when a MAC (port) sends to the PowerPC. Because we are modeling the MAC as having the same transfer characteristics as a Pentium, these values are the same as pt2pp_spp and pt2pp_spb.

pp2mc_spp, pp2mc_spb

Time on the secondary PCI bus when the PowerPC sends to a MAC (port). Because we are modeling the MAC as having the same transfer characteristics as a Pentium, these values are the same as pp2pt_spp and pp2pt_spb.

mp_ofact

Secondary PCI bus occupancy factor. This models the fact that the PCI bus utilization increases with additional bus masters. The secondary PCI bus has four bus masters. Based on early experiments, we use a factor of 0.7 so that the secondary PCI bus utilization is approximately the same as the per-port utilization for functions placed on the PowerPC. We did not use an occupancy factor for the primary PCI bus (for either the IXP1200 EEB or PMC694 models) because there are only two bus masters and our round trip measurements already took this into account.

pp_mc2_cpp, pp_mc2_cpb, pp_2mc_cpp, pp_2mc_cpb

The PowerPC cost to receive from and send to the ports. These are set to zero because we assume that the DMA operations are pipelined with the CPU cycles when we derived the value of 893 cpp. (See Section 3.3.2.)

`pp_pt2_cpp`, `pp_pt2_cpb`, `pp_2pt_cpp`, `pp_2pt_cpb`

The PowerPC cost to receive from and send to the Pentium. The per byte receive cost is zero because the Pentium pushes the data. The per byte send cost is zero because the DMA engine pushes the data. The per packet costs are set to be the same as StrongARM (DMA) case.

`pt_pp2_cpp`, `pt_pp2_cpb`, `pt_2pp_cpp`, `pt_2pp_cpb`

The Pentium cost to receive from and send to the PowerPC. The per byte receive cost is zero because the PowerPC pushes the data. Other costs are set to be the same as StrongARM case.

`pp_env_cpp`

The PowerPC environment cost in cycles/packet is set to be the same as the StrongARM.

`NUMPORTS`

The number of 100 Mbps ports on the PMC694. Set to 2.

Bibliography

[1] Alteon WebSystems, Inc., San Jose, California. *ACEnic Server-Optimized 10/100/1000 Mbps Ethernet Adapters Datasheet*, August 1999.

[2] F. Baker. Requirements for IP Version 4 Routers; RFC 1812. *Internet Request for Comments*, June 1995.

[3] A. Bavier, T. Voigt, M. Wawrzoniak, and L. Peterson. SILK: Scout Paths in the Linux Kernel. Technical Report 2002–009, Department of Information Technology, Uppsala University, Uppsala, Sweden, February 2002.

[4] J. C. R. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. In *Proceedings of the ACM SIGCOMM '96 Conference*, pages 143–156, Stanford, California, August 1996.

[5] M. A. Blumrich, R. D. Alpert, Y. Chen, D. W. Clark, S. N. Damianakis, C. Dubnicki, E. W. Felten, L. Iftode, K. Li, M. Martonosi, and R. A. Shillner. Design Choices in the SHRIMP System: An Empirical Study. In *Proceedings of the 25th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 330–341, Barcelona, Spain, June 1998.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[7] A. T. Campbell, S. Chou, M. E. Kounavis, and V. D. Stachtos. Implementing Routelets: Virtual Router Support for the IXP1200 Network Processor. In *Proceedings of the IXA University Program Workshop*, Portland, Oregon, June 2001.

[8] T. Chiueh and P. Pradhan. Suez: A Cluster-based Scalable Real-Time Packet Router. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)*, pages 136–143, Taipei, Taiwan, 2000.

[9] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0*, December 1997.

[10] P. Crowley, M. E. Fiuczynski, J.-L. Baer, and B. N. Bershad. Workloads for Programmable Network Interfaces. In *IEEE 2nd Annual Workshop on Workload Characterization*, Austin, Texas, October 1999. Also appears as Chapter 7 in [30].

[11] M. Dasen, G. Fankhauser, and B. Plattner. An Error Tolerant, Scalable Video Stream Encoding and Compression for Mobile Computing. In *Proceedings of the 1st Advanced Communications Technologies and Services (ACTS) Mobile Communication Summit*, pages 762–771, Granada, Spain, November 1996.

[12] B. Davie, August 1999. Personal Communication.

[13] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. *IEEE/ACM Transactions on Networking*, 8(1):2–15, February 2000.

[14] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small Forwarding Tables for Fast Routing Lookups. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 3–14, Cannes, France, October 1998.

[15] P. Druschel, L. L. Peterson, and B. S. Davie. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of SIGCOMM '94 Conference*, pages 2–13, London, October 1994.

[16] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. On Using Intelligent Network Interface Cards to support Multimedia Applications. In *Proceedings of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 95–98, Cambridge, UK, July 1998.

[17] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. SPINE: A Safe Programmable and Integrated Network Environment. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 7–12, Sintra, Portugal, September 1998.

[18] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.

[19] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. In *Proceedings of the 7th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 160–170, Cambridge, Massachusetts, October 1996.

[20] Y. Gottlieb and L. Peterson. A Comparative Study of Extensible Routers. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 51–62, New York City, June 2002.

[21] A. N. Habermann. *Introduction to Operating System Design*, pages 72–75. Science Research Associates, Inc., 1976.

[22] F. Hady, June 2002. Personal Communication.

[23] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93, Baltimore, Maryland, September 1998.

[24] IBM Microelectronics Division. *IBM PowerNP NP4GS3 Network Processor Solutions Product Overview*, April 2001.

[25] IEEE Standard 802.3. *Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*. IEEE, New York, NY, October 2000.

[26] InfiniBand Trade Association. *InfiniBand Architecture Specification, Release 1.0*, October 2000.

[27] Intel Corporation. *IXP1200 Network Processor Datasheet*, September 2000.

[28] Intel Corporation. *IXP12EB Intel IXP1200 Network Processor Ethernet Evaluation Kit Product Brief*, 2000.

[29] Intelligent I/O (I_2O) Special Interest Group. *Intelligent I/O (I_2O) Architecture Specification, Version 2.0*, March 1999.

[30] L. K. John and A. M. G. Maynard, editors. *Workload Characterization for Computer System Design*. Kluwer Academic Publishers, Boston, Massachusetts, March 2000.

[31] S. Karlin and L. Peterson. Maximum Packet Rates for Full-Duplex Ethernet. Technical Report TR–645–02, Princeton University, February 2002.

[32] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. *Computer Networks*, 38(3):277–293, February 2002.

[33] S. C. Karlin, D. W. Clark, and M. Martonosi. SurfBoard – A Hardware Performance Monitor for SHRIMP. Technical Report TR–596–99, Princeton University, March 1999.

[34] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[35] M. E. Kounavis, A. T. Campell, S. Chou, F. Modoux, J. Vicente, and H. Zhuang. The Genesis Kernel: A Programming System for Spawning Network Architectures. *IEEE Journal on Selected Areas in Communications*, 19(3):511–526, March 2001.

[36] F. Kuhns, J. DeHart, A. Kantawala, R. Keller, J. Lockwood, P. Pappu, D. Richards, D. Taylor, J. Parwatikar, E. Spitznagel, J. Turner, and K. Wong. Design of a High Performance Dynamically Extensible Router. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE 2002)*, pages 42–64, San Francisco, California, May 2002.

[37] T. V. Lakshman and D. Stiliadis. High Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching. In *Proceedings of the ACM SIGCOMM '98 Conference*, pages 203–214, Vancouver, British Columbia, Canada, September 1998.

[38] O.-I. Lepe-Aldama and J. García-Vidal. A Performance Model of a PC Based IP Software Router. In *Proceedings of the IEEE International Conference on Communications (ICC 2002)*, volume 2, pages 1230–1235, New York City, April 2002.

[39] O.-I. Lepe-Aldama and J. García-Vidal. I/O Bus Usage Control in PC-Based Software Routers. In *Proceedings of the 2nd International IFIP–TC6 Networking Conference (NETWORKING 2002)*, pages 1135–1140, Pisa, Italy, May 2002.

[40] M. Martonosi, S. Karlin, C. Liao, and D. W. Clark. Performance Monitoring Infrastructure in Shrimp Multicomputers. *International Journal of Parallel and Distributed Systems and Networks (Invited paper in the special issue on Measurement of Program and System Performance)*, 2(3):126–133, 1999.

[41] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, Seattle, Washington, October 1996.

[42] J. Moy. OSPF Version 2; RFC 2328. *Internet Request for Comments*, April 1998.

[43] C. Partridge. How Slow is One Gigabit Per Second? *ACM SIGCOMM Computer Communication Review*, 20(1):44–53, 1990.

[44] C. Partridge, P. P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. C. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. D. Troxel, D. Waitzman, and S. Winterble. A 50-Gb/s IP Router. *IEEE/ACM Transactions on Networking*, 6(3):237–247, June 1998.

[45] V. Paxson. Automated Packet Trace Analysis of TCP Implementations. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 167–179, Cannes, France, September 1997.

[46] PCI Special Interest Group, Hillsboro, Oregon. *PCI Local Bus Specification, Revision 2.2*, December 1998.

[47] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandelkar, A. Purtell, and J. Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, 19(3):473–487, March 2001.

[48] L. L. Peterson, S. C. Karlin, and K. Li. OS Support for General-Purpose Routers. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS–VII)*, pages 38–43, Rio Rico, Arizona, March 1999.

[49] J. Postel. Internet Protocol; RFC 791. *Internet Request for Comments*, September 1981.

[50] P. Pradhan and T. Chiueh. Operating System Support for Programmable Cluster-Based Internet Routers. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS–VII)*, pages 76–81, Rio Rico, Arizona, March 1999.

[51] X. Qie, A. Bavier, L. Peterson, and S. Karlin. Scheduling Computations on a Programmable Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, Cambridge, Massachusetts, June 2001.

[52] RAMiX Incorporated, Ventura, California. *PMC/CompactPCI Ethernet Controllers Product Family Data Sheet*, 1999.

[53] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP–4); RFC 1771. *Internet Request for Comments*, March 1995.

[54] N. Shalaby, L. Peterson, A. Bavier, Y. Gottlieb, S. Karlin, A. Nakao, X. Qie, T. Spalink, and M. Wawrzoniak. Extensible Routers for Active Networks. In *Proceedings of the DARPA Active Networks Conference and Exposition (DANCE 2002)*, pages 92–116, San Francisco, California, May 2002.

[55] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 216–229, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[56] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, February 1999.

[57] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast and Scalable Level Four Switching. In *Proceedings of the ACM SIGCOMM '98 Conference*, pages 191–202, Vancouver, British Columbia, Canada, September 1998.

[58] D. E. Taylor, J. S. Turner, and J. W. Lockwood. Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers. In *Proceedings of the 4th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 25–34, Anchorage, Alaska, April 2001.

[59] C. B. S. Traw and J. M. Smith. Hardware/Software Organization of a High-Performance ATM Host Interface. *IEEE Journal on Selected Areas in Communications (Special Issue on High Speed Computer/Network Interfaces)*, 11(2):240–253, 1993.

[60] Vitesse Semiconductor Corporation, Longmont, Colorado. *IQ2000 Network Processor Product Brief*, 2000.

[61] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 40–53, Copper Mountain Resort, Colorado, December 1995.

[62] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High Speed IP Routing Lookups. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 25–36, Cannes, France, October 1997.

[63] S. Walton, A. Hutton, and J. Touch. High-Speed Data Paths in Host-Based Routers. *IEEE Computer*, 31(11):46–52, November 1998.

[64] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 64–79, Kiawah Island Resort, South Carolina, December 1999.